Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik
Institut für Informatik
Lehrstuhl für Mensch-Computer-Interaktion

Enhancing Software Quality of Multimodal Interactive Systems

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften

der Fakultät für Mathematik und Informatik
der Julius–Maximilians–Universität Würzburg

vorgelegt von
Martin Walter Fischbach, M.Sc.

Betreuer: Prof. Dr. Marc Erich Latoschik

Acknowledgements

Der Weg ist das Ziel, sagte Konfuzius. Doch diesen Weg geht man nicht immer allein.

Liebste Anke, Du hast mich schon auf diesem Weg begleitet, bevor ich ihn bewusst begonnen hatte. Vielen Dank dafür, dass Du mich aufgefangen und immer wieder aufgerichtet hast. Dafür, dass Du mir Alternativen gezeigt und mich von anderen Pflichten entlastet hast, wenn es darauf ankam. Und dafür, dass ich mich bei Dir immer anlehnen und fallen lassen kann und konnte.

Lieber Marc, danke, dass Du mir dieses Forschungsfeld gezeigt hast und es mir ermöglicht hast meine eigenen Ideen zu verfolgen. Danke für Deine Anleitung und Motivation, aber auch für Deine Kritik die mich stets vorangebracht hat.

Lieber Dennis, ohne einen verlässlichen Mitstreiter und guten Freund wie Dich wäre ich an vielen Stellen nicht so weit gekommen. Danke für die vielen fruchtbaren Diskussionen, die großen und kleinen *Hackathons*, die Beteiligung an und Motivation zu Sport[1] und bewusster Ernährung sowie für Deine stetige Hilfsbereitschaft auch bei vielen *Quick Questions*.

Auf dem letzten Abschnitt des Weges wuchs die Zahl der Wegbegleiter. Lieber Chris und lieber Sascha, danke für die tollen gemeinsamen Projekte. Lieber Florian, lieber Jean-Luc, lieber Dominik, lieber Ecki, (und Chris,) danke für die geführten Diskussionen und Euer Feedback zu meinen Ideen. Liebe Sandra, lieber Kristof, lieber Eike, liebe Andrea, (liebe alle zuvor genannten,) liebe HCI-Gruppe, danke für das tolle Miteinander und die immerwährenden Prokrastinationsmöglichkeiten in der Kaffeeküche.

Zu guter Letzt möchte ich noch meinem kleinsten und wichtigsten neuen Wegbegleiter danken. Lieber Konrad, schön, dass Du da bist, mir Freude bereitest und mir – in Zeiten, in denen es mir unmöglich schien – gezeigt hast, dass es noch Wichtigeres gibt. Schön, dass es Dich gibt.

[1] danke auch an Simon, Chris und Kristof

Abstract

Multimodal interfaces (MMIs) are a promising human-computer interaction paradigm. They are feasible for a wide rang of environments, yet they are especially suited if interactions are spatially and temporally grounded with an environment in which the user is (physically) situated. Real-time interactive systems (RISs) are technical realizations for situated interaction environments, originating from application areas like virtual reality, mixed reality, human-robot interaction, and computer games. RISs include various dedicated processing-, simulation-, and rendering subsystems which collectively maintain a real-time simulation of a coherent application state. They thus fulfil the complex functional requirements of their application areas. Two contradicting principles determine the architecture of RISs: coupling and cohesion. On the one hand, RIS subsystems commonly use specific data structures for multiple purposes to guarantee performance and rely on close semantic and temporal coupling between each other to maintain consistency. This coupling is exacerbated if the integration of artificial intelligence (AI) methods is necessary, such as for realizing MMIs. On the other hand, software qualities like reusability and modifiability call for a decoupling of subsystems and architectural elements with single well-defined purposes, i.e., high cohesion. Systems predominantly favour performance and consistency over reusability and modifiability to handle this contradiction. They thus accept low maintainability in general and hindered scientific progress in the long-term.

This thesis presents six semantics-based techniques that extend the established entity-component system (ECS) pattern and pose a solution to this contradiction without sacrificing maintainability: semantic grounding, a semantic entity-component state, grounded actions, semantic queries, code from semantics, and decoupling by semantics. The extension solves the ECS pattern's runtime type deficit, improves component granularity, facilitates access to entity properties outside a subsystem's component association, incorporates a concept to semantically describe behavior as complement to the state representation, and enables compatibility even between RISs. The presented reference implementation *Simulator X* validates the feasibility of the six techniques and may be (re)used by other researchers due to its availability under an open-source licence. It includes a repertoire of common multimodal input processing steps that showcase the particular adequacy of the six techniques for such processing. The repertoire adds up to the integrated multimodal processing framework *miPro*, making *Simulator X* a RIS platform with explicit MMI support. The six semantics-based techniques as well as the reference implementation are validated by four expert reviews, multiple proof of concept prototypes, and two explorative studies. Informal insights gathered throughout the design and development supplement this assessment in the form of *lessons learned* meant to aid future development in the area.

Contents

Chapter 1

Introduction

> *The interaction of humans with their environment (including other humans) is naturally multimodal. We speak about, point at, and look at objects all at the same time. We also listen to the tone of a person's voice and look at a person's face and arm movements to find clues about his feelings. To get a better idea about what is going on around us, we look, listen, touch, and smell. When it comes to HCI, however, we usually use only one interface device at a time–typing, clicking the mouse button, speaking, or pointing with a magnetic wand. The "ease" with which this unimodal interaction allows us to convey our intent to the computer is far from satisfactory. An example of a situation when these limitations become evident is when we press the wrong key or when we have to navigate through a series of menus just to change an object's color.*
>
> *(Sharma, Pavlovic, and Huang, 1998)*

1.1 Multimodal Interaction

This criticism about Human-Computer Interaction (HCI) by Sharma et al. (1998) will soon be two decades old. Since then, the set of common sensors and rendering technologies as well as the available processing capabilities have been extended and improved due to technological advances. Prominent examples are smartphones, tablets, and even personal computers in combination with input devices for gesture detection, eye tracking, or markerless motion tracking. However, the essential of the quote still stays valid: interfaces are more oriented towards what can easily be detected, processed, and dealt with programmatically and less towards human capabilities. This is partly due to the fact that the human operator will utilize her capabilities to learn the interface usage and accept it anyway.

The fundamental idea of *MultiModal Interfaces* (MMIs) differs at this point. "*The inter-*

action of humans with their environment (including other humans) is naturally multimodal" (Sharma et al., 1998). Besides speech, humans use a variety of non-verbal channels or rather *modalities* for interpersonal communication: most importantly gestures, facial expressions, gaze, and body movements. MMIs aim to migrate their utilization to human-computer interaction. Two aspects of interpersonal communication are of special importance to this migration. Firstly, not all communication is performed intentionally, i.e., so that the sender is aware of the effect to the receiver in prior. While speech and gesture are the prime channels for intentional communication, eye movements and changes in body posture may let the communication partner draw conclusions on one's intentions and feelings. Secondly, communication is context dependent. Shared experiences, prior communication contents, and the surrounding environment have to be taken into account when interpreting communicated signals. MMIs thus have to use adequate modalities. For instance, speech and gestures to allow a user to intentionally give instructions to a system, supplemented by facial expressions and gaze to infer her mood without explicitly asking. In addition, the relevant context properties have to be made available for the computational analysis. By this means, current ways of using systems can be replaced with forms of interaction specific to the human communication capabilities (Latoschik, 2001b). This explicitly includes *multimodal input*, e.g., speech, gestures, and facial expressions, as well as *multimodal output*, e.g., visual display, audio, and tactile feedback.

A large variety of advantages are achievable this way. Sharma et al. (1998), Turk and Robertson (2000), Dumas, Lalanne, and Oviatt (2009), Oviatt (2012) summarize scientific contributions in the field and describe primary potential benefits: multimodal interfaces **increase expressiveness**, due to the fact that modalities can be used complementary to amplify, to modify, or to disambiguate. For instance, speaking and pointing at objects instead of describing referents solely with words, i.e. "*Give me [pointing] that bottle.*" vs. "*Give me the green bottle left of the book in the second shelf from the bottom.*". They **increase flexibility**, by utilizing the strengths of single modalities to compensate weaknesses of other. This is well exemplified by modern smartphones that offer a natural language user interfaces for web searches. In noisy environments or if it is inappropriate to speak, the user can switch to entering queries using a keyboard displayed on the smartphone's touch screen. They **increase reliability**, due to the fact that modalities can be used redundantly to convey information. In other words, considering multiple modalities from the same source—the human operator—reduces uncertainty for decision making (Murphy, 1996; Hall & Llinas, 1997). Altogether, these three advantages result in **increased efficiency**, when using multimodal interfaces.

Additionally, a wide range of further advantages are identified. It is easier to learn how to use multimodal interfaces, since they derive from human to human communication be-

havior and rely on modalities naturally used thereto. This contrasts with many traditional human-computer interfaces, where system provided in- and output concepts have to be accepted and learned by the user. MMIs aim to shift complexity to the computer system. As a consequence, they provide a preferably natural interface to the user and reduce the required cognitive load while interacting. The flexibility of multimodal interfaces can facilitate their use even if certain constraints or impairments exist. For instance, the opportunity to use a system with speech only, if the user is not able to use her hands. Altogether, ease of learning and flexibility result in a widened spectrum of possible users.

Oviatt and Cohen (2015) complement these arguments by highlighting the superior aptitude of multimodal interfaces for handling errors as well as their potential to stimulate cognition. The reasons for superior error handling are threefold on the user's side. Firstly, users tend to select modalities that are less error prone in the current situation or context. Secondly, the language supported by a multimodal interface is mostly simplified compared to the full extend of human communication. This is especially true for intentional instruction-based multimodal interfaces. Thirdly, users typically switch modalities after an error occurred, if possible. The potential to stimulate cognition is grounded on the higher expressive power of multimodal interfaces compared to keyboard interfaces or analogous non-digital tools, like pen and paper. For instance, Oviatt and Cohen (2015)'s studies targeting high-school learning show a variation in the communication behaviour of learning groups depending on the interface of the digital tool they utilize to support learning (keyboard- and mouse-based PC vs. pen- and speech-based tablet). The authors conclude that these findings demonstrate the direct relation beween language and thought.

1.2 Technical Realizations

MMIs are feasible for a wide range of environments, including traditional desktop and mobile scenarios. However, they are especially suitable if interactions have to be spatially and temporally grounded with an environment in which the user is (physically) situated. For instance, pointing at an object as part of a speech-gestural instruction is easier than providing a similar information using a keyboard, since it utilizes the natural spatial referencing of interpersonal communication. *Situated interaction environments* range from real (physical) spaces to so called *Virtual Environments* (VEs); respective application areas from smart homes and *Human-Robot Interaction* (HRI) to *Mixed Reality* (MR) and *Virtual Reality* (VR). MMIs are a promising alternative interaction technique for these environments (Latoschik, 2005; Ameri Ekhtiarabadi, Akan, Çürüklu, & Asplund, 2011; Cherubini, Passama, Fraisse, & Crosnier, 2015), which will become increasingly pervasive in our daily lives (Turk & Robert-

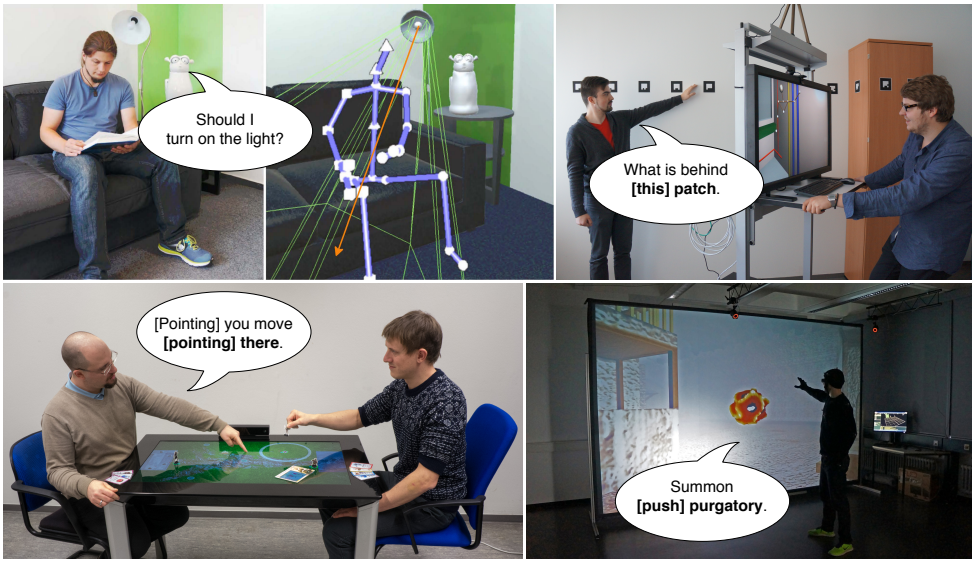

Figure 1.1. Examples of multimodal interfaces for situated interaction environments: a smart physical environment comprising a robot as assistant (upper left, image from Eckstein & B. Lugrin, 2016a), a collaborative augmented reality work station (upper right, image from Seufert, 2013), a mixed reality real-time strategy game (lower left, cf. Link et al., 2016), and a virtual reality spell casting game (lower right, cf. Fischbach et al., 2011). Terms in square brackets denote gestures that are executed in synchrony with the subsequent word(s).

son, 2000). Figure 1.1 illustrates MMIs for situated interaction environments, realized using contributions of this thesis.

Their prime distinguishing characteristic is the manner of feedback to the user. On the one end, feedback means altering the physical environment, for example moving a robot arm, displacing an object, or switching on a light. On the other end, VEs simulate either a replica of the real world or of a fantasy world and make the user belief that it is real by providing stimuli to her senses the same way the physical world does. The resulting feeling of being present in a non-real world is called *immersion*. The achieved level of immersion is a major aspect of VEs and depends, i.a., on the realization of interactions. In contrast to common interaction metaphors based on *Windows, Icons, Menus*, and a *Pointer* (WIMP), multimodal interfaces do not negatively influence immersion. To the contrary, they can even improve immersion in VEs (Latoschik, 2001b, 2005). Moreover, they facilitate natural communication with virtual agents and humanoid robots, e.g., enabling virtual agents as multimodal assistants (Kopp, Jung, Leßmann, & Wachsmuth, 2003).

In terms of their technical realizations, however, situated interaction environments do not differ that much. Respective software platforms fundamentally comprise means to capture

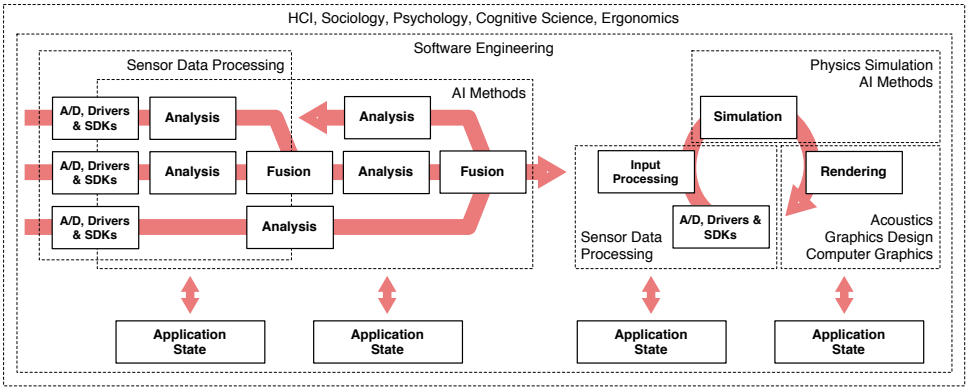

Figure 1.2. Subsystems (solid boxes) typically required to realize a MMI for a situated interaction environment. Involved research disciplines are emphasized using dashed boxes. Red arrows indicate communication and state access. One or more global application states are possible, depending on whether multimodal processing is done independently or is integrated into the RIS.

and process relevant user behavior and physical environment changes, simulate a virtual environment, and provide coherent rendering output to the user. They consist of multiple communicating subsystems that manage internal states and may also comprise global application states accessible by multiple subsystems. Performance is essential to the same extend as functional requirements to guarantee low system reaction times and hence usability, immersion, and security. Due to these requirements such systems are classified as *Real-time Interactive Systems* (RISs, cf. Wiebusch, 2016, pp. 6–8).

If complex input is to be analyzed by RISs or corresponding output to be generated, the integration of artificial intelligence (AI) methods as well as of a *knowledge representation layer* (KRL, Cavazza & Palmer, 2000) are fundamental requirements too. For instance, the need to handle symbolic concepts as well as relations between those concepts. RISs that fulfill those extra requirements are subcategorised as *Intelligent Realtime Interactive Systems* (IRISs, cf. Latoschik & Tramberend, 2011), respective VEs as *Intelligent Virtual Environments* (IVEs, Luck & Aylett, 2000). Their additional capabilities are a necessity to analyze and generate multimodal utterances, for instance input consisting of gestures and natural language. The processing of such input typically comprises modality individual analysis as well as combined analysis that finally derives a conjoint meaning. Figure 1.2 illustrates multimodal input processing in conjunction with a typical RIS architecture.

AI methods increase the complexity of subsystem interplay in addition to what is already implied due to the basic RIS simulation and -rendering functionality, i.e., interdependencies between subsystem-internal state representations, mutual state access, overall synchroniza-

tion, and flow of control. In combination with the necessary KRL, this poses high requirements to the system's architecture. However, if the user input to be analyzed reaches a certain complexity, AI methods and a KRL are inevitable for processing. MMIs for situated interaction environments are consequently amongst the highest demanding application areas for RISs. Yet, RISs are also applied for application areas with similar or lower requirements (Wingrave & LaViola, 2010), such as 3D computer games (typically called *game engines*), digital content creation or traditional Graphical User Interfaces (GUIs). Figure 1.3 gives an overview of (I)RIS requirements and application areas.

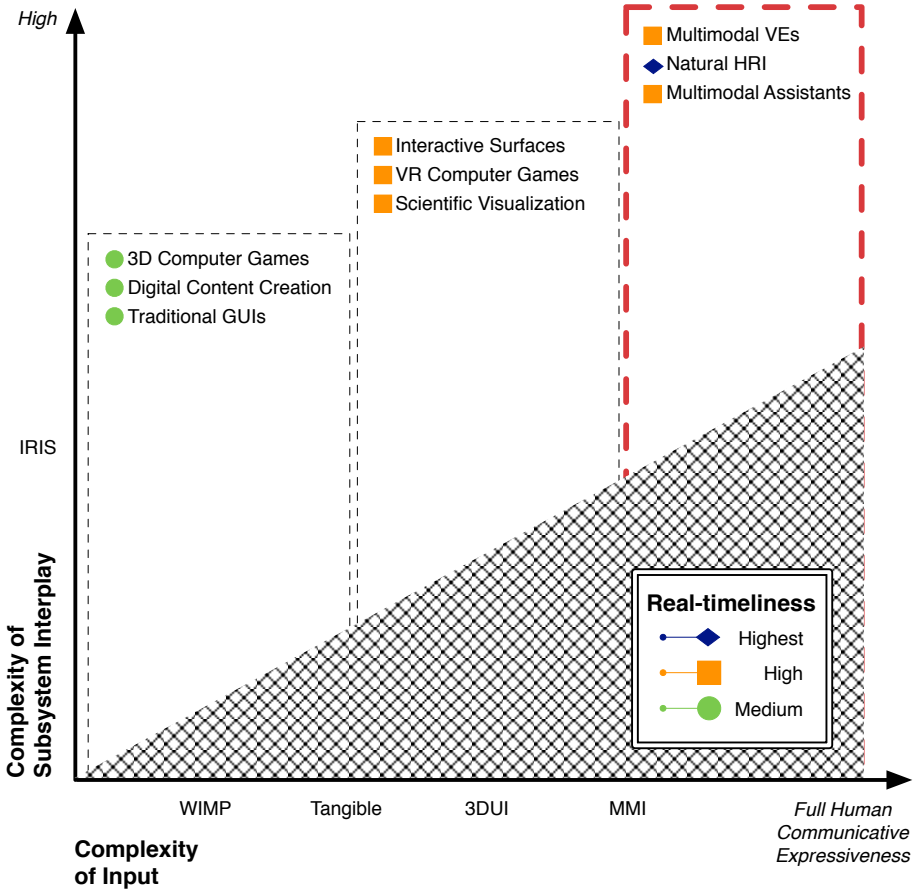

Figure 1.3. (I)RIS requirements broken down by complexity of input, complexity of subsystem interplay, and necessary real-timeliness. Dashed rectangles indicate a typical classification of three groups of RIS application areas with increasing overall requirements. Multimodal IRISs, the target area of this thesis (red), are characterized by high functional and real-time requirements as well as complex input. Refer to the text for details.

1.3 Problem Statement

Since Sutherland's (1964) visionary *Sketchpad*, new technical solutions for RISs have been introduced for decades (see Figure 1.4). Similarly, platforms and demonstrators for multimodal interfaces, so called *multimodal systems* (MMS), have been published since Bolt's (1980) *Put that there* (see Figure 1.5). In addition to these research systems, today commercial RISs (see Figure 1.6) are widely used in scientific projects for the development and evaluation of applications and interfaces. Due to their closed source, however, they are less useful for improving the software quality of RIS architectures.

Over the years, features as well as complexity have increased and different application areas have been explored—promoting multimodal interface software to a mature technology (Lalanne et al., 2009). This can be stated for RISs accordingly, since they are deployed both in large practical applications (e.g., RISs for computer games) and in the field of safety critical systems (e.g., RISs for robot control). This maturity, however, does not apply equally for all application areas and software quality aspects.

Most of the latest systems support non-functional software requirements to counter the negative impacts of ad-hoc tailored application-specific solutions. However, maintainability, comprising modularity, modifiability, and reusability, becomes more and more of an issue the higher the overall functional requirements get. In the area of multimodal interfaces for situated interaction environments it is a major issue (Latoschik, 2005; Steed, 2008; Lalanne et al., 2009; Latoschik & Tramberend, 2010; Latoschik & Fischbach, 2014; Fischbach, 2015). The main problem is a requirement contradiction known for some time as the *coupling dilemma* (Latoschik & Blach, 2008; Latoschik & Tramberend, 2010). On the one hand, low coupling of subsystems, in terms of their mutual access, is paramount to satisfy modifiability and reusability. This calls for abstract access to a common context representation, i.e. the application state. On the other hand, utilization of highly specific data structures, management of close temporal and semantic dependencies between many processing-, simulation-, and rendering steps, and the satisfaction of soft real-time constraints are major requirements too.

The severest consequences are low maintainability in the short term and, more importantly, hindered scientific progress in the long-term, due to limited repeatability and ability to build on previous results. In the commercial sector this circumstance is often compensated by high expenditures to redo major parts of software platforms that actually just required *minor* changes, e.g. a new 3D rendering subsystem. For the research community, which in most cases lacks the necessary resources for such an approach, the maintainability issue is highly problematic. In addition, poor *API usability* (Clarke, 2004; Daughtry, Farooq, Myers, & Stylos, 2009), comprising development effort and usability for developers, exacerbates

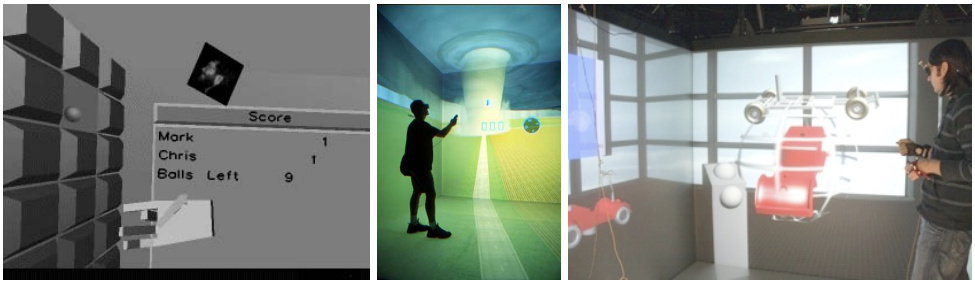

Figure 1.4. Research RISs over the last decades: *MR Toolkit* (left, image from Shaw & Green, 1993), *VR Juggler* (center, image from Cruz-Neira, Bierbaum, Hartling, Just, & Meinert, 2002), and *SCIVE* (right, image from Fröhlich, 2014).

Figure 1.5. Multimodal interaction research systems and demonstrators over the last decades: The *Put that there* demonstration (left, image from Bolt, 1980), the *Quickset* system (center, image from Cohen et al., 1997), and the *virtuelle Werkstatt* (right, image from Latoschik, 2005).

Figure 1.6. Current commercial RIS that are widely used in research: *Unreal Engine 4* (left, image from EPIC GAMES, INC., 2017) and *Unity 3D* (right, image from Unity Technologies, 2017).

reuse even more (Piccioni, Furia, & Meyer, 2013).

In the case of Latoschik's work, the integration of a multimodal processing framework into the RIS *AVANGO* (Tramberend, 1999) led to groundbreaking results, like the *virtuelle Werkstatt* and the multimodal interface to the virtual agent *Max* (Latoschik, 2005). However, the heavy dependency on AVANGO and its underlying, proprietary scene graph system (SGI's Performer) as well as the utilized scripting language, hindered long-lasting reusability (Kuck, Wind, Riege, Bogen, & Birlinghoven, 2008). It prohibited researchers to build upon these results, since the *virtuelle Werkstatt* has no currently running build or successor. Even worse, there is currently no RIS platform with explicit MMI support that is available for research (i.e., the source code can be obtained and if it is running on current hardware platforms) besides the contribution of this theses (cf. Fischbach, Wiebusch, & Latoschik, 2017).

The Entity-Component-System (ECS) pattern (Alatalo, 2011) has become a prominent approach to the coupling dilemma (e.g., used by Latoschik & Tramberend, 2011; Bueskens et al., 2014; Unity Technologies, 2017). This pattern organizes the data (the components) associated with subsystems (the systems) in an object-centered view (the entities) using composition over inheritance. This composition greatly enhances decoupling. Problems arise in cases where subsystems need mutual access to components outside of their primary data association. The major reason for this requirement is the need of AI subsystems to reflect the overall application state, e.g., to provide inference capabilities or to perform reference resolution during the analysis of a multimodal utterance. In fact, these issues do not only arise when realizing MMIs for situated interaction environments but with virtually any other IRIS subsystem combination. However, MMIs are particularly suited as use cases for developing solutions to the coupling dilemma, since they cover most of the its challenging issues: real-time state access to information ranging from low-level numeric to high-level semantic information (see Figure 1.2, left part) as well as adequate decoupling from the main simulation loop to not compromise the overall performance (see Figure 1.2, right part).

1.4 Objectives

Considering the pontential of multimodal interfaces, the large amount of RIS application areas, the coupling dilemma, and the lack of a research platform with explicit multimodal interface support, the following research question drives the efforts of this thesis:

> *What software techniques foster the maintainability of IRISs and also support the realization of multimodal interfaces for situated interaction environments?*
>
> *(revised from Fischbach, 2015)*

The investigation of this research question is subdivided into five objectives that are elaborated in the process of this thesis.

O_1 Analysis of (I)RIS maintainability issues, multimodal input processing requirements, and suitable evaluation methods.

O_2 Development of software techniques that counter the coupling dilemma.

O_3 Implementation of O_2, comprising typical multimodal input processing techniques.

O_4 Evaluation of O_2 and O_3 with regard to maintainability.

O_5 Implementation of proof-of-concept demonstrations accompanying the development.

Multimodal input is taken as primary use case for the development of solutions to the coupling dilemma. The resulting concepts are facilitating multimodal output generation too, since both fields heavily rely on the application of AI methods. Distinctive issues of multimodal output, however, are not addressed in this thesis.

1.5 Structure and Results

The thesis is divided into six remaining chapters subsequent to the *motivation and problem description* in this chapter. A detailed multimodal interaction use case is described in chapter 2 to support the analysis of issues, the identification of requirements, and the presentation of results later on. Related literature about (I)RISs, MMSs, and suitable maintainability evaluation methods is elaborated in chapter 3, followed by an *analysis* of the (I)RIS-MMS combination in chapter 4 to detail and undergird the problem description of this chapter and to further classify the contribution of this thesis (O_1). chapter 5 presents six *semantics-based software techniques* that address IRIS maintainability and jointly solve the coupling dilemma (O_2): semantic grounding, a semantic entity-component state, grounded actions, semantic queries, code from semantics, and decoupling by semantics. These techniques extend the established entity-component-system (ECS) pattern and overcome its main deficits with respect to state access. The *reference implementation* of the six techniques, comprising an integrated multimodal input processing framework, is showcased in chapter 6 by means of a walk-through of central implementation aspects and with the aid of the interaction use case (O_3). Withal, architectural design decisions and key implementation mechanisms that further facilitate maintainability are highlighted. At the end of this chapter, a brief overview of ancillary contributions to the reference implementation is given. Conducted *evaluations* (O_4) and *proof-of-concept* demonstrations (O_5) are described in chapter 7. The chapter includes a presentation of results obtained from the utilization of the reference implementa-

tion for *student projects* as well as for practical exercises in master level courses. chapter 8 concludes this thesis by summarizing and discussing achieved results and by pointing out potential future research benefiting for the presented contributions.

Most results of this thesis have been published at national as well as at international workshops, conferences, and journals. All relevant publications are listed in Table 1.1, each assigned to one or more aspects of this thesis that are adressed therein.

Table 1.1. Published results of this thesis, mapped to the following aspects addressed therein: Motivation and Problem Description (M&PD), Analysis (A), Semantics-based SoftWare Techniques (SSWT), Reference Implementation (RI), Evaluation (E), Proof-of-Concept (PoC), and Student Projects (SP).

Reference	Addressed Aspects
Fischbach, M., Wiebusch, D., & Latoschik, M. E. (2017, April). Semantic entity-component state management techniques to enhance software quality for multimodal VR-systems. *IEEE Transactions on Visualization and Computer Graphics*, *23*(4), 1342–1351	A, SSWT, RI
Fischbach, M., Wiebusch, D., & Latoschik, M. E. (2016, March). Semantics-based software techniques for maintainable multimodal input processing in real-time interactive systems. In *9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)* (pp. 1–6). IEEE Computer Society	A, SSWT, RI
Link, S., Barkschat, B., Zimmerer, C., Fischbach, M., Wiebusch, D., Lugrin, J. L., & Latoschik, M. E. (2016, March). An intelligent multimodal mixed reality real-time strategy game. In *2016 IEEE Virtual Reality (VR)* (pp. 223–224)	PoC, SP
Zimmerer, C., Fischbach, M., & Latoschik, M. E. (2016). Maintainable management and access of lexical knowledge for multimodal virtual reality interfaces. In *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology* (pp. 347–348). VRST '16. Munich, Germany: ACM	PoC, SP
Fischbach, M. (2015). Software techniques for multimodal input processing in realtime interactive systems. In *Proceedings of the 2015 ACM on International Conference on Multimodal Interaction* (pp. 623–627). ICMI '15. Seattle, Washington, USA: ACM	M&PD, A
Latoschik, M. E. & Fischbach, M. (2014). Engineering variance: software techniques for scalable, customizable, and reusable multimodal processing. In M. Kurosu (Ed.), *Human-computer interaction. theories, methods, and tools. hci 2014* (Vol. 8510, pp. 308–319). Lecture Notes in Computer Science. Cham: Springer International Publishing	SSWT, RI
Fischbach, M., Zimmerer, C., Giebler-Schubert, A., & Latoschik, M. E. (2014, September). [DEMO] Exploring multimodal interaction techniques for a mixed reality digital surface. In *2014 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)* (pp. 335–336)	PoC, SP
Zimmerer, C., Fischbach, M., & Latoschik, M. (2014). Fusion of mixed-reality tabletop and location-based applications for pervasive games. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces* (pp. 427–430). ITS '14. Dresden, Germany: ACM	PoC, SP
Giebler-Schubert, A., Zimmerer, C., Wedler, T., Fischbach, M., & Latoschik, M. E. (2013). Ein digitales Tabletop-Rollenspiel für Mixed-Reality-Interaktionstechniken. In *Virtuelle und Erweiterte Realität, 10. Workshop der GI-Fachgruppe VR/AR* (pp. 181–184). Informatik. Shaker Verlag	PoC, SP
Fischbach, M., Neff, M., Pelzer, I., Lugrin, J.-L., & Latoschik, M. E. (2013). Input device adequacy for multimodal and bimanual object manipulation in virtual environments. In *Virtuelle und Erweiterte Realität, 10. Workshop der GI-Fachgruppe VR/AR* (pp. 145–156). Informatik. Shaker Verlag	PoC, SP

Reference	**Addressed Aspects**
Fischbach, M., Treffs, C., Cyborra, D., Strehler, A., Wedler, T., Bruder, G., … & Steinicke, F. (2012). A mixed reality space for tangible user interaction. In *Virtuelle und Erweiterte Realität, 9. Workshop der GI-Fachgruppe VR/AR* (pp. 25–36). Informatik. Shaker Verlag	PoC
Fischbach, M., Wiebusch, D., Latoschik, M. E., Bruder, G., & Steinicke, F. (2012a). Blending real and virtual worlds using self-reflection and fiducials. In *Proceedings of the 11th International Conference on Entertainment Computing* (pp. 465–468). ICEC'12. Bremen, Germany: Springer-Verlag	PoC
Wiebusch, D., Fischbach, M., Latoschik, M. E., & Tramberend, H. (2012). Evaluating scala, actors, & ontologies for intelligent realtime interactive systems. In *Proceedings of the 18th ACM Symposium on Virtual Reality Software and Technology* (pp. 153–160). VRST '12. Toronto, Ontario, Canada: ACM	E, RI
Wiebusch, D., Fischbach, M., Strehler, A., Latoschik, M. E., Bruder, G., & Steinicke, F. (2012). Evaluation von Headtracking in interaktiven virtuellen Umgebungen auf Basis der Kinect. In *Virtuelle und Erweiterte Realität, 9. Workshop der GI-Fachgruppe VR/AR* (pp. 189–200). Informatik. Shaker Verlag	PoC
Fischbach, M., Wiebusch, D., Latoschik, M. E., Bruder, G., & Steinicke, F. (2012b). smARTbox A portable setup for intelligent interactive applications. In H. Reiterer & O. Deussen (Eds.), *Mensch & Computer 2012 — Workshopband: interaktiv informiert — allgegenwärtig und allumfassend!?* (pp. 521–524). München: Oldenbourg Verlag	PoC
Fischbach, M., Latoschik, M. E., Bruder, G., & Steinicke, F. (2012). smARTbox: out-of-the-box technologies for interactive art and exhibition. In *Proceedings of the 2012 virtual Reality International Conference* (19:1–19:7). VRIC '12. Laval, France: ACM	PoC
Fischbach, M., Wiebusch, D., Giebler-Schubert, A., Latoschik, M. E., Rehfeld, S., & Tramberend, H. (2011, March). Sixton's curse – Simulator X demonstration. In *2011 IEEE Virtual Reality Conference* (pp. 255–256)	PoC

Chapter 2

Use Cases

The basic demands described in the chapter 1 entail a set of further *fundamental requirements* that (I)RISs have to fulfill to support the implementation of common use cases. Since MMIs are a particularly suited as use cases for developing solutions to the coupling dilemma, an example interaction within such an interface is chosen to motivate those requirements and to point out the benefits of the presented approaches later on (revised from Fischbach et al., 2017):

A user furnishes a virtual room in an architectural modeling application via a speech- and gesture interaction (cf. Figure 2.1). At first she utters "*Put* [deictic gesture] *that green chair near* [deictic gesture] *this table*." (cf. Bolt, 1980) followed by "*Turn it* [kinemimic gesture] *this way*." (cf. Latoschik, 2002).

For the realization of this use case, the physical- and the virtual environment needs to be represented in the **application's state**. The physical environment comprises the user's communicative utterances, e.g., posture and spoken words. The virtual environment comprises objects like the table and the chair, including properties like positions, orientations, velocities, textures, and bounding boxes. Information about the physical environment is captured by sensors and provided via drivers or SDKs in different abstraction levels, e.g., positions and orientations from a tracking system or a text representation of spoken words from an *Automatic Speech Recognizer* (ASR). The application's state has to be **accessible** by all or at least be **communicated** to the relevant **system parts**. Those parts have to be **executed** to be able to process data, run simulations, and perform rendering that outputs a VE to the user(s). In order to handle the multimodal utterance, some of the sensor data may have to be processed before it can be used for a high-level combined analysis. For instance, meaningful gestures have to be determined from the positions and orientations of the user's joints over time or uttered words have to be annotated with their lexical category. The result of this processing is typically event-like symbolic data.

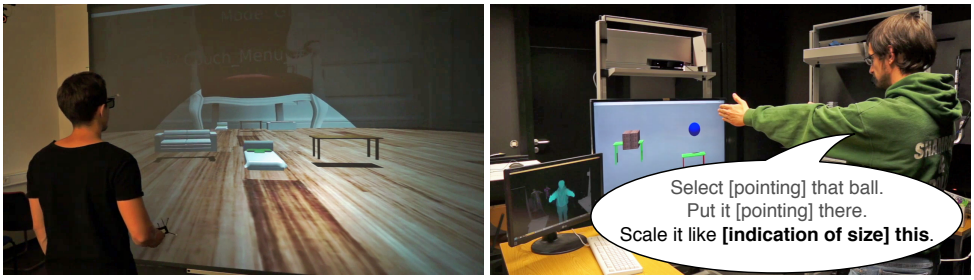

Figure 2.1. Users furnishing a virtual room: in a VR setup using ray-casting as well as a joystick for interaction (left, image from Kern, Kullman, Wiebusch, & Lugrin, 2016) and in a prototypical semi-immersive setup comprising a stereoscopic display using a speech and gesture interface (right, image from Kern, Kullman, Zöller, et al., 2016). Both demonstrations are results of practical master level courses, realized using contributions of this thesis. The movement analysis and continuous mapping to virtual object orientations, though, is not implemented to the full extend described by Latoschik (2002).

During multimodal analysis, is has to be checked if this data satisfies syntactic, temporal, and semantic constraints. **Syntactic correctness** involves checking of allowed successions, e.g., if a verb is followed by a noun phrase. **Temporal correctness** involves checking of co-occurrence within and between modalities, as in "[deictic gesture] *that*". **Semantic correctness** involves checking if the meaning derived by analyzing one modality conflicts with prior analysis, with other modalities, and with the context. For instance, if the chair is movable or if the object pointed at (gesture) really denotes a green chair (speech).

An analyzed utterance that proved valid has to result in an appropriate system **behavior** that is perceivable by the user. For example, the chair should appear next to the table or rotate as long as the user performs the respective gesture with her hand. In addition to deriving and executing an instruction, it is desirable to provide feedback to the user at the time of processing. This could include presenting intermediate results, like highlighting the green chair after the first part of the first example has been uttered.

For properly processing the second utterance, two further requirements have to be considered. Firstly, former application states have to be (partly) available. On a coarse time scale, this can be former utterances or rather the past discourse, e.g., the fact that the green chair has been selected in the first utterance, in order to resolve the anaphora *it* . On a fine time scale, this is necessary to foster temporal correctness in the presence of varying processing time of single multimodal channels. For instance, if the recognition of the pronoun *that* from the audio input takes longer than the calculation of the user's pointing direction from the latest tracking data, it is necessary to access former pointing direction values. In concrete terms, the pointing direction at the time *that* was uttered is has to be accessed to correctly deter-

mine the user's selection. Secondly, the realization of the kinemimic rotation gesture requires a continuous mapping of posture features to a virtual object, i.e. the chair's orientation.

Finally, all these functional requirements have to be fulfilled with respect to **performance**, including latency between user actions and system reactions as well as overall data throughput. A summary and specification of the motivated requirements is listed below, subdivided into requirements that apply for all RISs and requirements that are characteristic for IRISs as well as for multimodal input processing. Figure 2.2 supplements this summary by illustrating the requirements with the aid of the general system architecture presented in section 1.2.

The presented use case constitutes an instruction-based and intentional interface. It is based on well known contributions in the field to show how these familiar interactions can be implemented in a novel, efficient and maintainable way. However, the concepts developed therefrom are not constraint to the presented use case, since many requirements also hold for other interface types, like unintentional- or completely non-verbal ones. For instance, many of the required system capabilities and processing steps are equally necessary to process social signals. Mainly the final analysis step differs from speech-driven approaches.

RIS Requirements

R_{exe} System parts have to be executed to realize distinct aspects of the use case. These *subsystems* range from computationally light-weight threads of execution, e.g., for data processing, to computationally heavy-weight threads of execution, e.g., for simulation and rendering.

R_{com} Subsystems have to be able to communicate results to other subsystems.

R_{sta_1} As an extension to R_{com}, all virtual and physical objects that are relevant for more than one subsystem may be collectively represented in a uniform mannar. This collection of predominantly numeric data is called *application state*. Subsystems may manage additional internal states that are only relevant for their functioning.

R_{acc_1} If R_{sta_1} is fulfilled, all subsystems shall have access to the application state.

R_{per} The system shall react to changes of the relevant physical environment, by application state changes and subsequent rendering output, at least so fast and regularly that the application stays usable and secure. Ideally, end to end latency and jitter shall be so low that a further decrease does not improve usability or security.

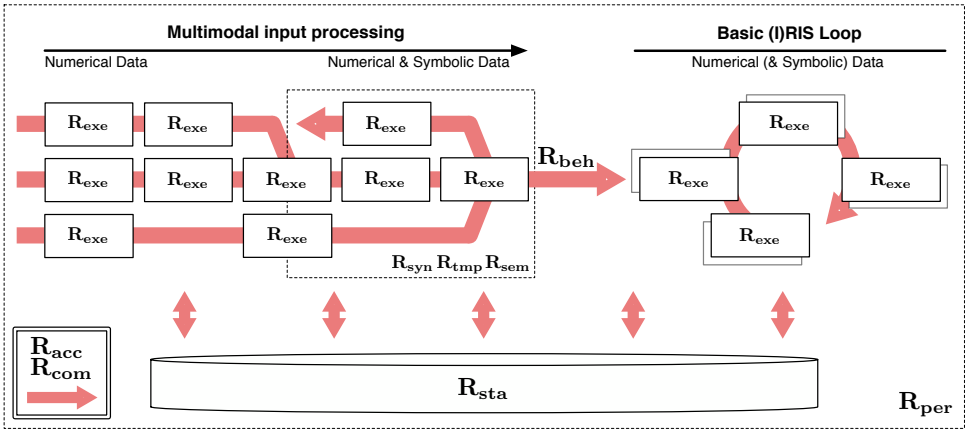

Figure 2.2. Fundamental requirements that IRIS have to fulfill to allow the implementation of typical MMI use cases. Refer to the text for details.

IRIS Requirements

R_{sta_2} In addition to R_{sta_1}, the application state shall also be able to represent symbolic data and potentially relations between elements of the state.

R_{acc_2} In addition to R_{acc_1}, subsystems shall be able to query the application state to access its elements based on their properties.

R_{beh} Operations that are triggerable by the user via the interface and that alter the application state in a way that can be perceived by the user, called *actions*, shall be represented so that (input analyzing) subsystems can occasion their execution. Ideally those subsystem can also query them and reflect on them.

Multimodal Input Processing Requirements

R_{syn} The system shall provide means to define syntactic structures that a succession of user input has to feature as well as means to validate input according to these definitions.

R_{tmp} The system shall provide means to define temporal dependencies that user input (of different modalities) has to feature as well as means to validate input according to these definitions.

R_{sem} The system shall provide means to define semantic constraints that a succession of user input and the current application state have to feature as well as means to validate input and application state according to these definitions.

Chapter 3

Related Work

In this chapter, relevant related work about MMSs, (I)RISs, as well as methods suitable to evaluate their maintainability is presented, analyzed and discussed. The chapter begins with the subject *maintainability* to be able to build on corresponding terminology and findings later on. Subsequently, terminology, applied methods, open challenges, and existing platforms in the area of MMSs and (I)RISs are presented and discussed. The chapter concludes with a summary that emphasizes the combination of MMS- and (I)RIS functionality.

3.1 Maintainability

Hard- and software environments change over the years and let software age (Parnas, 1994). Sensors, processing units, and output devices as well as operating systems, integrable frameworks, and libraries with superior functionality, efficiency, and effectiveness or lower price get available; old ones become no longer produced or supported. Similarly, requirements to and functional specifications of software systems change, due to opportunities created by new hard- and software, staff changes, or insights gained. This is especially true for the area of (I)RISs that heavily depends on a variety of complex hard- and software solutions and that is applicable in various disciplines, which potentially introduce ever new research questions and thus requirements for the underlying systems. Maintenance is thus essential if those systems are to be capitalized on in the long-term. The *maintainability* of a system is the "*degree of effectiveness and efficiency with which [... it] can be modified by the intended maintainers*" (ISO, 2011). It determines the resources and thus cost to keep a system useful. If these cost get too high, systems are stopped to be maintained and ultimately get unusable. A major problem, since researchers built on former achievements ever since.

According to the *ISO/IEC 25010:2011* standard (ISO, 2011), maintainability consists of the following sub-aspects:

Modularity *"Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components."*

Modifiability *"Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality."*

Reusability *"Degree to which an asset can be used in more than one system, or in building other assets."*

Analysability *"Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified."*

Testability *"Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met."*

With respect to MMSs and (I)RISs, modularity often corresponds to how the concerns of different simulation or analysis aspects can be separated, e.g., by a coarse subdivision of the system into subsystems, but also by more fine grained techniques and by such that affect orthogonal aspects like data representation or execution models. Modifiability includes adaptation to new hard- or software parts with minimal effort to adapt the rest of the platform and application. For instance, the possibility to exchange a visual rendering subsystem, as motivated in the introduction. Reusability includes the utilization of an (I)RIS or MMS for multiple applications, a typical trait of MMS- and (I)RIS platforms, but also the degree to which software assets can be used in or for other MMSs and (I)RISs. Analysability and testability concern the verification of the above qualities. Their level of difficulty is related to the complexity of the system to be verified.Moreover, they comprise the evaluation of system qualities beyond maintainability, e.g., performance aspects like latency and jitter.The latter aspects of analysability and testability, however, are not targeted by this thesis.

In addition to the sub-aspects defined by the *ISO/IEC 25010:2011* standard, the usability of a system's programming interface for developers, i.e., its *API usability* (McLellan, Roesler, Tempest, & Spinuzzi, 1998), plays a critical role in supporting maintainability (Clarke, 2004; Daughtry et al., 2009; Piccioni et al., 2013; Myers & Stylos, 2016). It includes an API's learnability, the efficiency and correctness with which a developer can use it, its quality to prevent errors, its consistency, and its matching to the developers' mental models. Most of these subaspects center around the provision of appropriate functionality and ways to access it, i.e, abstractions, affordances, and their perception by the developer. In addition to the concrete design of interfaces and documentation, tools can foster the usability of software platforms in general, e.g., autocompletion and refactoring support as part of development environments.

Assessment

Properly assessing or even comparing maintainability is a delicate endeavor with a limited amount of methods and no obvious choices (Riaz, Mendes, & Tempero, 2009). Objective maintainability assessment methods apply automated code analysis based on computable quality metrics (e.g., the *maintainability index* proposed by Oman & Hagemeister, 1992; Coleman, Ash, Lowther, & Oman, 1994). However, these approaches often exhibit drawbacks as being difficult to exploit for maintainability improvement (Heitlager, Kuipers, & Visser, 2007). For instance, if an automated evaluation's sole result is one number quantifying a system's maintainability, without revealing which sub-aspects should be revised and how this could be done. Despite efforts that are taken to counter this issue (e.g., by Heitlager et al., 2007), there is little evidence on their effectiveness (Riaz et al., 2009).

Subjective maintainability assessment methods are based on expert judgment in the context of reviews (Riaz et al., 2009; McIntosh, Kamei, Adams, & Hassan, 2016). Such *expert reviews* are the most commonly applied form of evaluation (Riaz et al., 2009). In addition to these predictive methods, maintainability can be assessed a posteriori, i.e., based on the maintenance activity.

The limitation in effective methods similarly applies for API usability (Piccioni et al., 2013; Myers & Stylos, 2016). Objective methods exist but are less commonly applied. For instance, automated evaluations using guidelines that identify function signatures with too many parameters of the same type in a row or that check the consistency of parameter orderings. Subjective methods are likewise the primary choice for API usability assessment. They comprise expert reviews based on (API usability) guidelines and user studies, such as think-aloud usability evaluation, API peer reviews (McLellan et al., 1998; Ruiz, Chen, & Oviatt, 2010) or questionnaires based on programming tasks (Piccioni et al., 2013). These assessment methods are typically supplemented with development processes that foster API usability in the first place (Myers & Stylos, 2016), e.g., by requirement elicitation methods (e.g., *natural programming*) or the use of guidelines. Moreover, commented code examples have proven beneficial for learning new APIs (McLellan et al., 1998).

Discussion

Despite the importance of maintainability and API usability, conducted evaluations for MMSs and (I)RISs primarily focus functional subsystem qualities, performance and user interface usability (see section 3.2 and section 3.3). The lack of effective methods in this area is additionally exacerbated by the complexity of the systems to be evaluated (Wiebusch, 2016; Fischbach et al., 2017). All subjective measures exhibit one crucial requirement: develop-

ers that have sufficient knowledge of a system to solve complex tasks and to reflect on non-functional qualities. Getting a reasonable amount of such experts is problematic. At least, for widespread commercial systems the availability of experts is moderate. Yet, those systems are typically restrictive when it comes to profound system modifications, necessary to conduct research on maintainability. Open-source research systems possess such flexibility, however, they are often used by only few developers.

Given these constraints, the most reasonable method to foster maintainability as well as API usability are expert reviews (cf. Steed, 2008; Kuck et al., 2008). They require a low number of developers compared to other subjective methods and yield insights for actually improving a system. Moreover, they can be iteratively applied to identify and correct deficits. This primary method can be supplemented by objective measures as well as by studies involving developers that are system novices but available in higher numbers, e.g., master-level students attending a dedicated course. Finally, tool support can be explored to implement elicited requirements or identified issues concerning API usability.

3.2 Multimodal Systems

The use case presented in chapter 2 showcases what kind of user interface is desirable and which difficulties its technical realizations will likely have to face. Since software quality is the main aspect of the motivated problem, systems dedicated to the realization of MMIs are consequently researched first. This section presents an overview of MMSs, beginning with terms and concepts used in literature as well as applied methods. Subsequently, challenges are summarized and concrete solutions, i.e., software platforms, are listed and discussed. For comprehensive presentations of multimodal interfaces and -systems, targeting aspects alternative to the software quality focus in this thesis, you may refer to the following publications:

- **Multimodal Interfaces in General**
  Sharma et al., 1998; Jaimes and Sebe, 2007; Turk, 2014; Oviatt and Cohen, 2015
- **Motivation**
  Turk and Robertson, 2000; Oviatt, Coulston, and Lunsford, 2004
- **Principles, Models and Systems**
  Dumas, Lalanne, and Oviatt, 2009
- **Fusion**
  Lalanne et al., 2009; Atrey, Hossain, El Saddik, and Kankanhalli, 2010

Terminology

This section specifies the main terms related to MMIs relevant for this thesis. Besides MMI itself, these are: *modalities*, the communication channels MMIs rely on, *multimodal systems*, technical realizations of MMIs, and *multimodal fusion*, the process of jointly analysing modalities.

Modality

One of the most central terms for MMIs is *modality*. There are several points of view on what a modality can be. Johnston et al. (1997) describe it in general as "*... [a channel] through which information may pass between user and computer*". Jaimes and Sebe (2007) view modality from a human-centered perspective and concretize their definition by giving concrete examples:

> "*We use a human-centered approach and by modality we mean mode of communication according to human senses and computer input devices activated by humans or measuring human qualities (e.g., blood pressure). The human senses are sight, touch, hearing, smell, and taste. The input modalities of many computer input devices can be considered to correspond to human senses: cameras (sight), haptic sensors (touch), microphones (hearing), olfactory (smell), and even taste. Many other computer input devices activated by humans, however, can be considered to correspond to a combination of human senses, or to none at all: keyboard, mouse, writing tablet, motion input (e.g., the device itself is moved for interaction), galvanic skin response, and other biometric sensors.*"

The authors supplement their definition by emphasizing a human- and a system perspective:

> "*[...] as we type, we touch keys on a keyboard to input data into the computer, but some of us also use sight to read what we type or to locate the proper keys to be pressed. Therefore, it is important to keep in mind the differences between what the human is doing and what the system is actually receiving as input during interaction.*"

In analogy to Johnston, Cohen, McGee, Oviatt, Pittman, and Smith's (1997) definition, the channel through which information passes is essential. Such channels link, on the one end, the human as actor and, on the other end, the computer as receiver. They are fittingly called *human-action modalities* and *computer-sensing modalities* by Sharma et al. (1998). The

authors conclude that there is a large numer of potential (human-action) modalities that can be considered for MMIs, since computers can sense human qualities that humans themselves can not:

> "*[... It] is desirable that computers be able to interpret all natural human actions. Hence, computers should interpret human hand, body, and facial gestures, human speech, eye gaze, etc. Some computer-sensory modalities are analogous to human ones. Computer vision and ASR mimic the equivalent human sensing modalities. However, computers also possess sensory modalities that humans lack. They can accurately estimate the position of the human hand through magnetic sensors and measure subtle changes of the electric activity in the human brain, for instance. Thus, there is a vast repertoire of human-action modalities that can potentially be perceived by a computer.*"

In the remainder of this thesis, the term *modality* refers to *human-action modalities* to remain with the concept of migrating human-human interaction.

Multimodal Interface

A *multimodal interface* can thus be defined as an interface that "*[... aims] to recognize naturally occurring forms of human language and behavior*" (Oviatt, 2003) and that "*[... supports] input and processing of two or more modalities [...]*" (Oviatt & Cohen, 2015). Symmetrically, multimodal interfaces may also comprise *multimodal output*, i.e., may "*[use] different modalities, like visual display, audio, and tactile feedback, to engage human perceptual, cognitive, and communication skills in understanding what is being presented.*" (Turk & Robertson, 2000). The research fields associated to those two aspects of multimodal interfaces share many methods and approaches, however, they also have to cope with distinct issues.

Multimodal System

A *multimodal system*, in turn, is considered as a computer system that can be interacted with via a multimodal interface. If the affinity to natural human behavior is omitted and a multimodal system is seen as "*[a system] that responds to inputs in more than one modality or communication channel*" (Jaimes & Sebe, 2007), a wider range of technical realizations fit the definition; even a keyboard and mouse interface. However, the manner of modality utilization is essential for a further classification. Nigay and Coutaz (1993) propose to classify MMIs by means of three dimension: the *levels of* data *abstraction*, the temporal *use of modalities*, and the presence of a combined analysis of modalities (*fusion*). In this design space the first

example interaction presented in chapter 2 ("*Put* [deictic gesture] *that green chair near* [deictic gesture] *this table*") would be classified as follows: The modalities speech and gestures are used in *parallel*. In terms of data abstraction, raw numerical values as well as symbolic values have to be processed. For instance, positions of the user's hand over time for the detection of deictic gestures (numeric) and recognized spoken words and detected gestures (symbolic). The modalities require a *combined* analysis to facilitate the resolution of the sentence's subject and object, i.e., the pointing direction of the user has to be evaluated with respect to the time the corresponding pronoun has been uttered. A multimodal system supporting such an interaction is called *synergistic*. In the remainder of this thesis, the terms multimodal interface and multimodal system refer to the synergistic use of modalities and to an affinity to natural human behavior. This choice demands the highest system requirements of the design space, whose solutions are certainly also suitable for less complex interfaces.

Coutaz et al. (1995) supplement MMI classification by considering goals that a user can reach within an application, e.g., causing an object to be selected or altering one of its properties to a specific value. The types of modalities that she can utilize to reach a certain goal as well as their temporal relationship is used to characterize an interface. Their proposed scheme is called the *CARE* properties (Complementarity, Assignment, Redundancy, and Equivalence). Modalities can be *equivalent* if only one out of multiple choices is sufficient to reach a certain goal. In contrast, one modality can be *assigned* if it is the only possible choice to reach a certain goal. Modalities can be used *redundantly* if they are equivalent and are nevertheless used sequentially or in parallel within a certain time window. Finally, modalities can be used in a *complementary* manner if they are used sequentially or in parallel within a certain time window, but one of them alone would not be sufficient. That is, if the necessary information is communicated divided amongst different modalities.

Both commands presented in chapter 2 require a complementary use of modalities, since neither speech nor gestures could be skipped. In fact, complementarity is the dominant theme for interactions in MMIs if users are left the choice, rather than redundancy (Oviatt & Cohen, 2015). Its utilization in principle has obvious advantages. Information can be passed using the most adequate modality for the user, e.g., speech for communicating actions, object types as well as their properties and gestures for communicating spatial information. A complementary use of modalities demands a synergistic multimodal system and thus entails the highest system requirements.

Multimodal fusion

In order to realize synergistic multimodal systems, the input modalities have to be jointly analyzed at some point of processing to derive a conjoint meaning *"most likely expressed by*

the user" (Pfleger, 2004). Lalanne et al. (2009) give a general definition that lists commonly used terms for the process:

> "*The mechanisms used for combining information (whether it is received in a sequential or parallel way) have received different names in the past. [This process is called] combining [... ,] cooperation of modalities [... ,] integration [... ,] multimodal integration [..., or] fusion.*"

In the remainder of this thesis the term *multimodal fusion* will be used, since it is most widely used, especially amongst the lastet publications in the field.

Dimensions of Fusion

Fusion can be applied on various levels of abstraction. Three *levels of fusion* have been identified throughout the literature (Hall & Llinas, 1997; Sharma et al., 1998; Dumas, Lalanne, & Oviatt, 2009; Hoste, Dumas, & Signer, 2011): *data-level fusion*, *feature-level fusion*, and *decision-level fusion* (see Figure 3.1). This classification scheme is further refined by the term *semantic-level fusion*, denoting fusion methods at decision-level that explicitly combine (preprocessed) semantic information from multiple modalities, which are loosely coupled with respect to their temporal occurrence (Oviatt & Cohen, 2015). For instance, the final processing required for the speech-gestural utterances of the interaction use case.

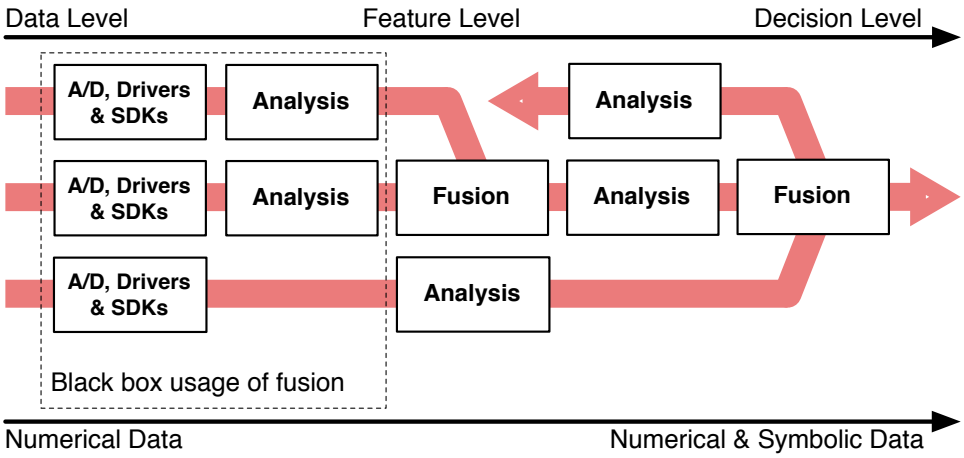

Figure 3.1. Levels of fusion in multimodal input processing. MMSs potentially apply fusion at data-, feature-, and decision level. Data-level fusion is oftentimes used as black box (dashed box), e.g., within SDKs or an integrated ASR subsystem. Fusion applied at feature-level, decision-level, or in-between, is typically realized using the MMS's architecture to access requirements and to communicate results (red arrows, cf. Jaimes & Sebe, 2007).

The crucial part of a multimodal system realization is the fusion of information that is obtained via multiple modalities into a semantically and temporally compatible interpretation. This process evaluates the captured user behavior with regard to syntactic, temporal, and semantic correctness (R_{syn}, R_{tmp}, and R_{sem}) to derive the user's intention. Atrey et al. (2010), Oviatt and Cohen (2015) propose three central dimensions of multimodal fusion methods to structure the variety of approaches that can be applied: *when to fuse*, *what to fuse*, and *how to fuse*. They are supplemented with the issue *how to specify* and detailed in the remainder of this section:

When to fuse refers to the levels of fusion presented in Table 3.1. "*Finding an optimal fusion level for a particular combination of modalities is not straightforward*", though (Sharma et al., 1998).

What to fuse complements the levels of fusion with appropriate data structures as shown in Table 3.2. While data- and feature-level fusion gets along with modality-specific data structures, decision-level fusion requires uniform representation that can be used for all modalities (R_{sta}).

How to fuse refers to concrete methods that can be applied for fusion. Theoretical foundations underlying many of those methods, e.g., the general fusion model proposed by Sharma et al. (1998), typically consider psychological and biological findings, such as neurological models, evidence accruement, contextual dependency (cf. Stein & Meredith, 1993), or methods to deal with discordance (cf. Bower, 1974). Atrey et al. (2010) present a comprehensive overview of feature-level fusion methods, categorized into rule-based methods, classification-based methods, and estimation-based methods. Amongst them, classification-based meth-

Table 3.1. Potential fusion levels in MMSs, clarified by listing conditions for their application, characteristics, and examples (cf. Sharma, Pavlovic, & Huang, 1998).

Level	Conditions	Characteristics	Example
Data	• Observations of the same type • High level of synchronization	• Sensitive to noise • Direct use uncommon in MMS • More likely used within drivers or SDKs	• Multiple cameras capturing visual information on one object
Feature	• Preceding analysis • Medium level of synchronization	• Possibly large feature sets • Common for MMS	• Speech and lip movement
Decision	• Preceding mode decisions	• Very common for MMS • Prone to information loss on lower levels • Robust to noise	• "*Make* [pointing] *this box white*"

Table 3.2. Appropriate data representations for fusion levels, clarified by listing commonly used data types as well as examples (cf. Jaimes & Sebe, 2007).

Level	Data	Common data structures	Example(s)
Data	• Raw sensor values	• Floating point vectors	• Pixel intensities • Audio levels
Feature	• Processed sensor values	• Floating point vectors • Strings	• Color histograms • Positions, Velocities • Audio features • Raw speech tokens
Decision	• Unimodal classification results • Results of data- or feature-level fusion	• Frames • Feature structures • Typed feature structures	• Grounded speech tokens • Classified gestures • Classified facial actions

ods, like neural networks, hidden markov models, and support vector machines, are most commonly applied. In addition to their utilization for feature-level fusion, many of those methods can also be applied for unimodal classification (Sharma et al., 1998). Decision-level fusion methods found in literature can be categorized into *statistical approaches* (e.g., Wu, Oviatt, & Cohen, 1999), approaches based on *frames and (typed) feature structures* (e.g., Cohen et al., 1997), *unification* (e.g., Johnston et al., 1997; Lukas, Schwägerl, & Latoschik, 2010), *finite-state transducers* (e.g., Johnston & Bangalore, 2000; Bangalore & Johnston, 2009), *temporal Augmented Transition Network*s (tATNs, e.g., Latoschik, 2002), and machine-learning-based approaches (e.g., Ngiam et al., 2011; Martínez & Yannakakis, 2014). An outline of data-level fusion methods is omitted, since they are commonly utilized as black box within drivers and SDKs, e.g., corresponding to ASRs and tracking systems.

The choice of a fusion method implies a certain functionality that has to be realized within a system. Yet, this choice does not directly influence a systems general maintainability. Concrete implementations rather effect intra-subsystem qualities. Indirectly, however, the necessitated state access of these methods is a potential source for close coupling. Especially decision-level fusion methods typically require to reflect on potentially all application state elements to validate semantic correctness (R_{sem}). In the context of the use case, for instance, to check if one of the objects pointed at is really a green chair.

How to specify refers to the configuration of fusion methods. At least at decision-level, a distinction between valid utterances that shall trigger a certain reaction by the application and invalid input is necessary. Machine-learning based approaches require annotated test data. Other approaches require some kind of (formal) definition of valid multimodal utterances and preferably a mapping to actions of the respective application. Utilized definitions of

that kind range from idiosyncratic specifications in programming code, formatted text, and domain specific languages (DSLs) to general multimodal interaction modeling languages, like the Multimodal Integration Markup Language (MIML, Latoschik, 2002), the Synchronized Multimodal User Interaction Modeling Language (SMUIML, Dumas, Lalanne, & Ingold, 2010), or XMMVR2 (Olmedo, Escudero, & Cardenoso, 2008). Dumas, Lalanne, and Oviatt (2009), Dumas et al. (2010) provide comprehensive overviews. As a guideline for the development of a multimodal interaction modeling language, Sire and Chatty (2004) propose ideal language features as the result of a requirements analysis. According to these authors, a language shall

1. be modality agnostic,
2. provide a binding mechanism to link actions of the respective application,
3. provide explicit control structures,
4. provide extensible event definition mechanisms,
5. provide a sophisticated data modeling, and
6. consist of reusable components.

While formality and reusability is typically high for languages that use common external formats, like XML, the call for explicit control structures and for a binding mechanism to application reactions especially benefits internal (in-code) definition languages.

Challenges

With this repertoire of methods at hand it is possible to analyze information from multiple modalities and also combine them into a common representation. Concrete implementations, i.e., multimodal systems, have to deal with a variety of challenges in doing so (Sharma et al., 1998; Lalanne et al., 2009; Atrey et al., 2010; Turk, 2014; Fischbach, 2015). On the technical side, multimodal fusion implies the processing and communication of data (R_{exe} and R_{com}), with typical characteristics that a system has to deal with:

Timestamps Every chunk of data has to be associated with the time it was communicated by the user (to be able to verify temporal constraints—R_{tmp}).

N-best results SDKs and processing steps often do not provide one reliable option (out of several) about what the user expressed at a certain point in time, but n-best guesses.

Confidences Items of n-best results are provided with a measure that describes the certainty that the represented information really complies with what the user expressed.

Formats The formats of data obtained via different modalities usually differs, especially at data-level. It has to be fused into a compatible representation at lest at decision-level.

Rates The rates with which data is provided to a multimodal system by sensors or rather drivers usually differs between modalities. This implies the requirement of suitable processing capabilities (R_{exe}) as well as of sufficient performance (R_{per}).

In addition, system architectures have to cope with:

Context Fusion methods have be able to access current and past properties of the captured physical environment (including user behavior) as well as of the virtual environment that are relevant for the user's multimodal utterances (R_{acc}, cf. *multimodal memory* Kopp, Bergmann, & Kahl, 2013; Bergmann, Kahl, & Kopp, 2014).

Processing times The analysis and fusion of different modalities may take substantially varying amounts of time (R_{exe}).

Parallelism A concurrent execution model is required to fully exploit moden hardware architectures, i.e., multiple cores and processing units (R_{exe}).

Asynchrony In addition to disparate capture rates, libraries that communicate with sensor hardware via drivers may have to run asynchronously to each other (R_{exe}).

Finally, the realization of the fusion process itself entails challenges, like which modality to capture, which features to extract, and which fusion method to apply (corresponding to the design issues presented above).

Despite the stated maturity of the multimodal interface domain (Lalanne et al., 2009) that may apply for certain sub-areas, other researcher conclude that "*[...] the field is still young*" (Dumas, Lalanne, & Oviatt, 2009). Pending grand challenges range from the improvement of the fusion process itself (Jaimes & Sebe, 2007; Lalanne et al., 2009; Dumas, Lalanne, & Oviatt, 2009; Latoschik & Fischbach, 2014), e.g., in terms of flexibility to cope with individual user characteristics and cultural context, and software quality issues of MMSs (Sharma et al., 1998; Turk & Robertson, 2000; Latoschik, 2005; Jaimes & Sebe, 2007; Lalanne et al., 2009; Dumas, Lalanne, & Oviatt, 2009; Latoschik & Tramberend, 2010; Latoschik & Fischbach, 2014; Fischbach, 2015) to the qualitative comparison of fusion methods (Sharma et al., 1998; Lalanne et al., 2009; Atrey et al., 2010; Fischbach, 2015) and considerations *beyond multimodality* (Oviatt & Cohen, 2015), like the influence of MMIs on the users' cognition. Amongst those, software quality issues of MMSs are of particular importance for this thesis. In the following sections, this challenge will be detailed, be means of a elaboration on MMS- and RIS platforms and continued with an analysis of their combination.

Platforms

In this section, MMS platform architectures are reviewed with respect to the following aspects: applied engineering techniques, identification of maintainability issues, and conduct of respective evaluations. This review completes the requirements elicitation for MMS and constitutes a basis for analysing the combination of MMS and RIS. Table 3.3 gives an overview of MMSs and shows which aspects are explicitly emphasized in the reference(s). The fact that an applied engineering technique, an identified maintainability issues, or a conducted evaluation is not emphasized, does not exclude that it was not performed or considered by the authors. Not being emphasized in the main system presentation, however, reflects significance. Moreover, if a system's source code is not available to other researchers, these presentations remain the only source of reference.

Bolt's (1980) *Put That There* demonstration is often referred to as the first system that showcased the technical feasibility of an interface that uses natural human-action modalities, i.e., speech and deictic gestures. *Demonstrations* that followed present different application areas, modalities, as well as improved processing and performance capabilities. Their commonality is the dedication for a specific application and a fixed set of subsystems.

Pioneered by systems like *Quickset* and *virtuelle Werkstatt* non-functional software qualities like reusability, modifiability, and modularity as well as development effort then grew important. The result of this development are *domain dependent multimodal system platforms* that provide different sets of methods for multimodal processing *and* the necessary means for the implementation of applications in one specific application area, e.g., desktop or mobile applications, interactive surfaces, virtual environments, or robot control. These systems consequently include subsystems required for realizing interactive applications, i.e., subsystems for simulation as well as for rendering, and usually provide means to develop applications. The concrete repertoire of available subsystems and particularly considered non-functional qualities, like performance or security, are aligned with typical application area requirements.

Finally, beginning with *OpenInterface*, another type of system emerged: *independent multimodal system platforms*. These systems aim to further increase maintainability, in particular the reusability of multimodal processing and fusion methods that are implemented as part of such systems, by not dedicating to a specific application area. That is, by omitting means for simulation and rendering.

Software engineering techniques that are applied to realize the listed MMSs and that are explicitly emphasized in the associated publication(s) can be categorized into: separation of concerns, communication schemes, execution models, state representation models, and state

access schemes (see Table 3.3, column *Techniques*). These categories in principle correspond to the basic RIS requirements identified in chapter 2, which shows the similarity between RIS- and MMS platforms. In fact, domain dependent MMS platforms overlap with RIS platforms, since they provide means to implement interactive applications, i.e., to process multimodal user input, to simulate an internal state, and to provide feedback to the user.

Separation of Concerns

- Component models, e.g., in *ICARE/FACET*, *OpenInterface*, and *SSI*
- Modularization, e.g., in *Cubricon*, *i*Chameleon*, and *HCI^2*
- Software agents, e.g., in *Quickset*, *Meanings4Fusion*, and *HephaisTK*

Execution Models

- The actor model in *miPro*
- Data flow graphs in *virtuelle Werkstatt* (attribute sequences)

Communication Schemes

- Events, e.g., in *SSI*, *Mudra*, and *OpenInterface*
- Messages, in *Quickset* and *miPro*
- Pipelines, in *SKEMMI* and *ICARE/FACET*
- Publish/subscribe mechanisms, in *HCI^2*
- Routes, in *virtuelle Werkstatt*
- Streams, e.g., in *SSI*, *OpenInterface*, and *Cubricon*

State Representation Models

- Fact bases, in *Mudra*
- Organization graphs and functional-semantic structures, in *eXpert TRAnslator*
- OWL, in *Meanings4Fusion*
- Semantic entities, in *virtuelle Werkstatt*
- A semantic entity-component state (an extension of semantic entities), in *miPro*
- Typed feature structures, in *Quickset*

State Access Schemes

Explicitly through

- Interface definitions, in *ICARE/FACET*
- Semantic queries, in *miPro*

Implicitly through

- Graphical tools, in *OpenInterface* and *SKEMMI*
- High-level description languages, e.g., in *SSI*, *COLD*, and *virtuelle Werkstatt*
- Rules, in *M3I* and *Mudra*

Besides the early demonstrations, all of the related publications identify at least one aspect of maintainability as a crucial system requirement, mostly driven by the aim to support the reuse of once implemented processing and fusion techniques (see Table 3.3, column *Issues*). All systems apply separation of concerns to foster modularity. In addition, the concretely applied software techniques influence the maintainability of the MMS platform.

In order to foster maintainability and in particular to reduce the development effort for the development of MMIs, many authors emphasize the importance of API usability. The main means applied are (graphical) tool support and (XML-based) high-level description languages, including multimodal grammars (e.g., Latoschik, 2005; Dumas, Lalanne, & Ingold, 2009), rules (e.g., Hoste et al., 2011; Möller, Diewald, Roalter, & Kranz, 2014), and internal DSLs (e.g., Latoschik & Fischbach, 2014; Fischbach et al., 2017). Both of which allow to (implicitly) define access to the application state.

A less identified, but equally important system quality is its capability to support methods on all processing- and fusion levels (Hoste et al., 2011; Fischbach et al., 2017). Hoste, Dumas, and Signer's (2011) approach to achieve unified multimodal processing builds upon a central fact base, a declarative rule-based description language, and an inference engine. The approach presented in Fischbach et al. (2017) and this thesis make use of an entity-based application state, a uniform semantic access scheme to it, as well as of the actor model.

Besides proof of concept applications, an evaluation of non-functional system qualities is rarely presented in literature, though (see Table 3.3, column *Evaluation*). Performance measures and expert reviews are the most reasonable applied methods. In addition, pre-studies and informal evaluations at least aim for getting a better intuition of achieved system qualities. This confirms the lack of suitable evaluation methods identified in section 3.1.

Discussion

Altogether, the prime achievements gathered by contributions over the years are abstract findings like software engineering techniques, methods, algorithms, and even best practices or lessons learned. Yet, it would be highly beneficial if any (old) system in showcased in literature could still be used (see Table 3.3, column *Available*). Seven of the presented MMS platforms are available for researches, which is owed to good maintainability as well as expended effort and ressources to maintain the systems. However, very few publications provide insight in the utilized execution model, a central system architecture characteristic, which affects maintainability. In addition, most systems present state representation models that are dedicated to decision-level fusion, leaving out that a real system somehow has to cope with low-level numerical sensor data and potentially even with properties of VE objects as well cf., Hoste, Dumas, and Signer's (2011). In combination with that lack of evaluations, this leaves little scope to comprehend and improve maintainability issues. When it comes to application areas that pose high system demands themselves, like situated interaction environments, available solutions are rare, as discussed in section 4.1.

Table 3.3. Overview of MMSs categorized into **demo**nstrations, **dom**ain dependent system platforms, and **ind**ependent system platforms(extended from Fischbach et al., 2017). System aspects explicitly emphasized in the reference(s) are summarized by identified issues (**Modu**larity, **Modi**fiability, **Reus(e)**ability, & **API u**sability), applied engineering techniques (**Exe**cution models, **Com**munication schemes, **Sta**te representation models, state **Acc**ess schemes, and supportive **Tool**s), and conducted evaluations (**Exp**ert reviews, **Pre**-studies, **P**roof **o**f **C**oncepts, & **Inf**ormal evaluations). Explicit *RIS support* is subdivided into support for VEs and RC. The systems' availability for research is indicated if the source code can be obtained and if it is running on current hardware platforms (see Appendix A for details and chapter 4 for *).

Name	Type	Emphasized system aspects														RIS support	Available	Reference
		Issues				Techniques						Evaluation						
		Modu	Modi	Ruse	APIu	SoC	Exe	Com	Sta	Acc	Tool	Exp	Pre	PoC	Inf			
Put that There	demo													x		no	no	Bolt (1980)
Cubricon	demo					x		x						x		no	no	Neal, Thielman, Dobes, Haller, and Shapiro (1989)
eXpert TRAnslator	demo					x			x					x		no	no	Wahlster (1991)
ICONIC	demo					x								x		yes (VE)	no	Koons and Sparrell (1994)
QuickSet	dom.		x		x	x		x	x							no	no	Cohen et al. (1997)
SGIM & virtuelle Werkstatt	dom.	x	x	x	x	x	x	x	x			x	x	x		yes	no*	Latoschik (2001a, 2005)
OpenInterface (OI)	ind.			x	x	x		x		x	x			x		-	yes	Serrano et al. (2008)
SKEMMI (OI)	ind.			x	x			x		x	x			x	x	-	yes	Lawson, Al-Akkad, Vanderdonckt, and Macq (2009)
Meanings4Fusion (OI)	ind.	x		x		x		x	x	x	x			x		-	yes	Mendonça, Lawson, Vybornova, Macq, and Vanderdonckt (2009)
i*Chameleon	ind.	x	x	x	x	x		x			x			x	x	-	no	Tang et al. (2011)
Mudra	ind.			x		x		x	x	x	x			x		-	no	Hoste et al. (2011)
unnamed system using COLD	dom.			x	x					x				x		yes (RC)	no	Ameri Ekhtiarabadi et al. (2011)
HCI^2	ind.	x	x	x	x	x		x		x	x			x		-	yes	Shen and Pantic (2013)
SSI	ind.	x		x		x		x		x				x		-	yes	Wagner et al. (2013)
M3I	dom.				x	x		x		x				x		no	yes	Möller et al. (2014)
unnamed system	dom.			x										x		yes (RC)	no	Cherubini et al. (2015)
miPro (Simulator X)	dom.	x	x	x	x	x	x	x	x	x	x	x	x†	x	x	yes (VE)	yes	Latoschik and Fischbach (2014), Fischbach et al. (2017), † see chapter 7

3.3 Real-time Interactive Systems

The second essential aspect of the interaction use case presented in chapter 2 is the environment in which it takes place. In contrast to traditional HCIs, the user is—in this case virtually—situated within the interface; not in front of it. Objects that the user can see (possibly even hear or touch) can be used to communicate commands and intentions to the system, e.g., they can be named, described or pointed at from the user's current perspective. The technical realization of situated interaction environments is highly complex Steed (2008), Wingrave and LaViola (2010), Wiebusch (2016), even without considering a MMI.

This section thus elaborates on RISs that constitute technical realizations of situated interaction environments. In analogy to the previous section, terms and concepts used in the RIS domain are presented first, followed by a brief summary of applied methods as well as open challenges, and a final listing and discussion of existing systems. For a comprehensive analysis of (I)RIS architectures with respect to reusability, you may refer to Wiebusch (2016).

Terminology

Virtual Environment

The characterization of a user's surroundings during (human-computer) interaction is typical for the domain of RISs. Two of the earliest established terms are *virtual environment* (VE, Fisher, McGreevy, Humphries, & Robinett, 1987) and *virtual reality* (VR, Lanier, 1988), denoting a completely virtual surrounding. Milgram, Takemura, Utsumi, and Kishino (1995) suggest a taxonomy for mixing real and virtual words and define a VE (a.k.a *VR environment*)

> "*The commonly held view of a VR environment is one in which the [participant] is totally immersed in a completely synthetic world, which may or may not mimic the properties of a real-world environment, either existing or fictional, but which may also exceed the bounds of physical reality by creating a world in which the physical laws governing gravity, time and material properties no longer hold.*"

Although the authors focussed on display technology for classification, which disregards input and other potential output modalities, they were the first to propose a unified perspective on VEs and other, partially- or non-virtual, HCI environments:

> "*In contrast [to VEs], a strictly real-world environment clearly must be constrained by the laws of physics. Rather than regarding the two concepts simply as antitheses, however, it is more convenient to view them as lying at opposite ends of a continuum, which we refer to as the Reality-Virtuality (RV) continuum.*"

Mixed Reality

All conceptual environments between a completely real- and virtual world are denoted as *mixed reality* (MR) according to Milgram et al. (1995). MR includes two sub-concepts distinguished by the surrounding environment observed by the user: principally real environments, enhanced with virtual artefacts, are called *augmented reality* (AR) and principally virtual environments, augmented with physical artifacts, called *augmented virtuality* (AV). Characteristics features of MR environments can be adopted from Azuma's (1997) later definition of AR:

> "*This survey defines AR as any system that has the following three characteristics:*
>
> 1. *Combines real and virtual*
> 2. *Interactive in real time*
> 3. *Registered in 3-D*"

The demanded real-timelines as well as the spatial registration between real- and virtual content are essential requirements for situated 3D- and multimodal interaction techniques too. Thus, MR environments are per se suited for such interactions.

Situated interaction environments

With the aim of emphasizing environments that share the common characteristic of being suitable for a MMI, the term *situated interaction environment* is used in this thesis. It denotes environments in which the user is (physically) situated and that thus foster interactions that are spatially and temporally grounded therein. Situated interaction environments comprise VEs, MR environments, and physical environments and can be characterized by

1. being completely real (as in HRI),
   combining real and virtual artefacts (as in MR),
   or being completely virtual (as in VE)
2. being interactive in real-time
3. fostering interactions that are spatially and temporally grounded

The term builds on Milgram, Takemura, Utsumi, and Kishino's (1995) and Azuma's (1997) definitions, emphasizes environments rather than systems or technologies, and explicitly includes environments that only allow physical feedback (e.g., moving a robot arm, displacing an object, or switching on a light). It relates to the concept of *situated interaction* (cf. Schmidt, Van de Velde, & Kortuem, 2000; Streitz, Röcker, Prante, Stenzel, & van Alphen, 2003; Bohus,

2014), which is more oriented towards physical environments, and adopts its use for VEs, e.g., to describe the interaction with a virtual agent (cf. Kopp et al., 2003; Leßmann, Kopp, & Wachsmuth, 2006; Salem, Kopp, Wachsmuth, & Joublin, 2010).

Intelligent Virtual Environment

Another source of complexity for VEs different to the user interface can be its content. Luck and Aylett (2000) propose the term *intelligent virtual environment* (IVE) as a

> "*This combination [AI, artificial life, VR, and VEs] of intelligent techniques and tools, embodied in autonomous creatures and agents, together with effective means for their graphical representation and interaction of various kinds, has given rise to a new area at their meeting point, which we call intelligent virtual environments.*"

Although the authors focus on autonomous agents, the implied requirements of IVEs mostly overlap with those of with those of multimodal input processing: integration of AI methods and elaborated knowledge representation. Moreover, more complex VEs may also call for adequate interaction techniques, making IVEs implicitly a target for MMIs.

Real-time Interactive System

As the counterpart to environments and concepts, *Real-time interactive systems* (RISs) denote their (software) technical realizations. While the term RIS also includes concrete applications and demonstrators, a RIS platform characterizes a generalized et of functionality dedicated to the development of multiple applications within an application area. RIS platforms provide means for the execution of multiple subsystems as well as for communication between subsystems. Optionally they provide a uniform data representation layer for all subsystems. They meet—some more or less strict—real-time constraints and are dedicated to the processing of user input, the simulation of an internal state, and the provision of feedback to the user. Typically RIS platforms already comprise implementations of commonly required subsystems, e.g., driver and SDK integrations for sensors and input devices, physics simulation, and visual rendering. They allow the technical realization of situated interaction environments, including VR and MR environments, as well as of scenarios with lower requirements (cf. Figure 1.3 in section 1.2).

Substantial contributions to (I)RIS software architectures have been encouraged by the IEEE Virtual Reality workshop on Software Engineering and Architectures for Realtime Interactive Systems (Latoschik, 2017, SEARIS). In its call for participation, the workshop describes RIS as (listing AR on one level with MR, in contrast to Milgram et al., 1995):

> "*[... RIS] span from Virtual Reality (VR), Augmented Reality (AR), and Mixed Reality (MR) environments to novel Human-Computer Interaction systems (such as multimodal or multitouch architectures) and entertainment applications in general. Their common principle is a strong user centric orientation which requires real-time processing of simulation aspects as well as input/output events according to perceptual constraints.*"

Intelligent Real-time Interactive System

Intelligent Real-time Interactive Systems (IRISs) are a RIS subcategory and the technical realization counterpart to IVEs, i.e., RISs that explicitly support the integration of AI methods and provide elaborated knowledge representation, e.g., capable of handling symbolic and numeric data. An alternative definition is given by Luck and Aylett (2000), which complements their autonomous agents-focused IVE definition:

> "*In this work, the more comprehensive term Intelligent Realtime Interactive System (IRIS) is used to denote a system that simulates a virtual environment the elements of which support their utilization in intelligent ways. In this regard, an element can be any part of the VE, including software modules that perform the necessary simulations.*"

Summary

The commonly used terminology in the RIS area overlaps in many aspects. Situated interaction environment is a a conceptual hypernym for physical environments, the MR spectrum (including AR), and for VEs. It promotes MMIs as promising human-computer interfcae and is a basis for immersion in VR and MR. Depending on their content, situated interaction environments can also be denoted as IVEs, e.g., if they comprise autonomous agents.

RISs are technical solutions for situated interaction environments. If they enable the realization of MMIs, i.e., explicitly support the integration of AI methods and provide elaborated knowledge representation, they are denoted as IRISs. (I)RIS platforms are thus the primary focus of this thesis. The presented use case is chosen to be within a VE. An IRIS that technically realizes it, however, should likewise suitable for realizing other forms of situated interaction environments (cf. section 7.2).

Architecture Concepts

RIS architectures have to lay the foundation for the fulfillment of the fundamental requirements identified in chapter 2. R_{exe} calls for the (parallel) execution of threads of execution, ideally on all levels of granularity, i.e., for an execution model for submodules. R_{sta} and R_{acc} demand data representation and sharing. This can be realized by a state representation that is mutually shared and accessible by all subsystems (global application state). At a minimum, this can also be realized by providing a communication scheme (R_{com}) between subsystems and by leaving the task of state representation to the subsystem developer (internal state). A communication scheme is required in both cases, since access to a global shared application state requires communication. Potential solutions to the above named demands have to be in line with the performance requirement (R_{per}) of RISs and should be in line with the maintainability requirement motivated by this thesis. Maintainability can typically not be pinned down to one specific engineering technique, however, its subaspect modularity is fostered by separation of concerns in almost any system. These general categories match those of MMS platforms identified in section 3.2, since both, MMSs and RISs, realize interactive applications. Differences potentially emerge in their concrete realization.

Wiebusch (2016, pp. 33–37) identifies the following (partly overlapping) common architecture concepts: component-based-, graph-based-, and message-based architectures as well as event systems, entity models, and the entity-component system (ECS) pattern. Concrete systems oftentimes cannot be clearly categorized, since they mix multiple concepts. Yet, the concepts can be mapped to execution models (exe), communication schemes (com), state representation models (rep), state access schemes (acc), and separation of concerns (soc).

Component-based architectures partition threads of execution into *components* by means of object-oriented programming constructs (soc). They do not define an execution scheme.

Graph-based architectures partition threads of execution into *nodes* (soc). Nodes can pass information to other nodes through *routes* (com) that connect *fields*, i.e, in- and output ports of nodes. Nodes can react on field value changes or be explicitly triggered. The propagation of field values through routes thus implicitly executes all behavior (exe), however, concrete field propagation strategies may vary.

Message-based architectures support the transfer of messages between separate threads of execution (com). Messages have to be addressed to one or more specific receivers. A broadcast to every possible receiver may be supported. Message-based architectures do not define an explicit execution scheme. They are a suitable basis for other architecture concepts and techniques, e.g., for graph-based architectures (realizing communication between node ports), for event systems, or for client/server architectures.

Event systems are similar to message-based architectures since they support the transfer of information between separate threads of execution (com). In contrast to messages, events do not have to be addressed to specific receivers. They are emitted by event producers. Potential receivers have the necessary means to indicate (request) that they are interested in notifications upon the emission of certain events. Such an emission then triggers a reaction defined by the receiver. This can imply a central registry for event propositions and request, which redistributes messages or conducts initial hand-shaking. Event systems do not define an explicit execution scheme.

Entity models are concepts that allow uniform state representation without relying on or extending a data structure that is dedicated to a specific simulation or rendering aspect, in contrast to, e.g., scene graphs. The whole state to be represented is composed of entities, "*[...] a "thing" which can be distinctly identified*" (Chen, 1976). An entity, in turn, is composed of its basic characteristics (Mannuß, Hinkenjann, & Maiero, 2008), i.e., is "*a set of [variables] that represent its properties*" (Wiebusch, 2016, p. 160). Examples of entities are virtual objects that can be (visually) perceived by the user, but also input device- and user representations or even subsystem configurations.

The **entity-component system** pattern is an extended entity model (rep) that also provides a separation strategy for threads of execution. Threads of execution are called *systems*. Entities aggregate *components*, which comprise one or more entity property that systems need for realizing a certain functionality aspect Alatalo, 2011; Wiebusch, 2016. Systems and entities are thus related via components, as they define on which entities a system operates and how it can access the entity (acc). A component could be an entity's aspect to be placeable, comprising properties like position and orientation, or its aspect to be a rigid body, comprising properties like mass and velocity. The ECS pattern originates from computer game architectures (Bilas , Scott, 2002; Orchard, Leslie, 2013; West, Mick, 2007) and is consequently used in many game engines (e.g., Unity Technologies, 2017; EPIC GAMES, INC., 2017).

None of these concepts cover all system requirements, so they have to be combined or supplemented with additional techniques, e.g., processes as execution model. Some of the concepts complement well, e.g., an entity model and the ECS pattern or an event system implemented based on messages. The first two constitute a subsystem agnostic and uniform state representation approach. They are thus superior to approaches relying on central and highly optimized—but subsystem dependant—models, like extended scene graphs (Kuck et al., 2008; Latoschik & Blach, 2008). The latter two are beneficial for decoupling the execution of subsystems and thus for the implementation of distribution capabilities (Steed, 2008).

What is left to realize are semantic dependencies between subsystems (Latoschik & Tramberend, 2010). For instance, what a specific message or event denotes (including its payload),

what properties an entity can have, and what they represent. Ultimately, this comes down to interface definitions, i.e., an agreement on names and meaning for classes, functions, variables, etc., that systems have to specify.

Challenges

Concrete RIS architectures are combinations of these concepts, partly including idiosyncratic solutions. Their development is typically targeting one more or less narrowly scoped application area that determines requirement details. The design, implementation, and maintenance of RIS, however, is accompanied by a number of challenges that apply to most systems. Steed (2008), Wingrave and LaViola (2010), Taylor et al. (2010) reflect on several years of RIS development and summarize common issues, best practices, and lessons learned:

> RISs necessitate high update rates to stay interactive, maintain immersion or registration with the physical world, and thus guarantee usability and security for the user. This performance requirement conflicts with the typically complex, functional RIS requirements.
>
> Application development and evaluation challenges range from efficient content creation over interface design issues, including usability support, user behaviour prediction, as well as the establishment of 3D interface metaphors and guidelines, to the proper capturing of information relevant to conducted experiments. A consequence, especially of the design issues, is that RISs require a high amount of iterative prototyping.
>
> Development effort can be countered by supporting developers with appropriate tools, processes and abstraction layers, i.e., by fostering API usability. Aspects highlighted by the surveys are learnability, complexity of configuration and building, support of common formats, and integration of common libraries.

In addition to its usefulness for realizing APIs, abstraction is also fundamental for a RIS architecture itself. It fosters decoupling thus modifiability and reusability. However, the reuse of (parts of) applications and subsystems, respectively their substitution, is a pressing issue (Ponder, Papagiannakis, Molet, Magnenat-Thalmann, & Thalmann, 2003; Wiebusch, 2016). The motivated endeavor for decoupling is opposed to the close temporal- and semantic coupling required to achieve performance and to provide a coherent simulation (Latoschik & Blach, 2008; Latoschik & Tramberend, 2010; Fischbach et al., 2017). The resulting maintainability issues are exacerbated by sparsely spread standards that foster the exchange of solutions amongst the community. Maintaining RISs is associated with high costs. These costs

are undertaken nevertheless, due to the potential of these systems. Yet, they negatively effect the durability of system and shorten the time until a system is no longer used. Steed (2008) summarizes reasons from over 15 years of RIS development for why his group stopped using a system:

1. The system was perceived as being hard to program or lacking in capability, suggesting a move to a similar, but potentially more powerful system
2. The system was retired by the authors
3. Knowledge was lost (usually by the owner graduating)
4. The system was no longer up to scratch (e.g. visually)
5. Lack of a critical facility (e.g. no cluster support)
6. Hardware or operating system support was no longer available

All of these reasons are consequences of low maintainability (1.–3.) or have not been resolved due to high maintenance costs (4.–6.).

Altogether, the implementation of a RIS is not straightforward (Kuck et al., 2008; Steed, 2008; Wingrave & LaViola, 2010; Taylor et al., 2010). RISs are complex, e.g., in terms of the interconnections between subsystems, sometimes even chaotic due to hidden or unpredicted dependencies. developers require a variety of skills from different research disciplines as well as intelligence and experience to be able to understand solutions and analyze malfunction. The grand challenge for RIS developers is to handle this complexity Ponder et al., 2003.

Platforms

Many software engineering techniques are at the disposal to realize fundamental RIS requirements and about as many reference implementations have been presented for validation as well as for practical application. In this section, RIS platform architectures of the last three decades are reviewed (in the same mannar as MMS) with respect to the following aspects: applied engineering techniques, identification of maintainability issues, and conduct of respective evaluations. Table 3.4 gives an overview of RISs and shows which aspects are explicitly emphasized in the reference(s).

Similar to the historic development of MMSs, early contributions were rather *demonstrations* than platforms with a fixed set of subsystems tailored for one specific application. Their main goal was to explore feasibility. Nevertheless, they (and all other systems) apply separation of concerns to foster modularity.

Systems like *MR Toolkit* and *DIVE* where then the first to shift the focus to the required development effort, as one aspect of API usability, and to the modifiability of the systems. Later system presentations emphasized the effort to reuse software solutions, e.g., subsystems, in different system configurations or even in different systems. These *RIS platforms* provide fundamental functionality, like execution schemes, communication patterns, and data representation models. They include a set of subsystem implementations that can be extended by application developers. They are thus suited for the development of various applications, in principle not limited to one area. A second principal feature of many RIS platforms is the distribution of computing load to multiple threads, CPUs, or cluster nodes (Allard et al., 2004), for instance implemented by *MR Toolkit*, *DIVE*, *Avocado*, *VR Juggler*, *FlowVR*, and *AvangoNG*. Especially the early systems depended on this technique to cope with the high computational demands of VR and MR applications. The technical advance in CPU and GPU hardware reduced the need of inter-node distribution. However, intra-node distribution is a requirement of any RIS today and inter-node distribution is still necessary for special hardware setups, like CAVEs, and computationally demanding application areas, like scientific simulation.

IRIS publications again shifted the focus to elaborated data and behavior representation models as well as access schemes that are capable of uniformly covering any aspect of the simulated environment and of the system. Besides numerical low-level properties commonly represented in RIS, this especially comprises the high-level semantics of (virtual) objects and actions relevant for an application. Two main use cases can be identified: artificial agents (*SCIVE*, *ISReal*, *REVE*, and *MASCARET*) and multimodal processing (*SCIVE* and *Simulator X*). The first use case requires semantic information to allow artificial agents for the reflection of the environment it perceives, e.g., to derive possible actions it can take. The latter use case requires this information to be able to check semantic correctness during the analysis of a user utterance (R_{sem}). IRISs certainly apply elaborated execution schemes and communication patterns, however, most contributions do not detail on them, which may stem from that these requirements are more sufficiently researched with enough suitable solutions available.

In addition to (I)RISs published by the research community, commercial RISs lately gained popularity amongst researches that use RISs in the role of application or subsystem developers (e.g. Latoschik et al., 2016; J. L. Lugrin, Zilch, Roth, Bente, & Latoschik, 2016). These systems typically offer functionally rich implementations of common subsystem types (especially high quality visual renering), a high amount of ready-to-use assets, high performance optimization, as well as low development effort due to good tool support and documentation. These advantages are made possible by a high amount of human ressources a succesful company is able to invest; in contrast to common scientific scenarios. On the downside,

commercial RIS focus on the demands of the market, primarily the computer game industry, and are protected by strict licences (EPIC GAMES, INC., 2017) or are close-source altogether (Unity Technologies, 2017). The first drawback manifests in commercial RISs being rather conservative about novel techniques and methods—from a researcher's point of view—since they have to guarantee reliability. If a feature desired for research was not anticipated by a RIS, it is oftentimes inefficient or impossible to add it properly; rendering the basically low development effort irrelevant. The latter drawback either hinders the publication of results (due to licence restrictions) or prevents the propper reflection of system behavior that is potentially influencing the research conducted with the system (due to restricted code access).

In terms of software quality, almost all RIS publications identify at least one aspect of maintainability as a crucial system requirement (see Table 3.4, column *Issues*). The reduction of development effort is mentioned most frequently, followed by reusability. All systems apply separation of concerns to foster modularity. Further engineering techniques that are emphasized in the references of Table 3.4 can be categorized into: communication schemes, execution models, state representation models, and state access schemes (in analogy to the MMS overview of Table 3.3).

Software engineering techniques that are applied to realize the listed RISs and that are explicitly emphasized in the associated publication(s) can be categorized into: separation of concerns, communication schemes, execution models, state representation models, and state access schemes (see Table 3.4, column *Techniques*). These categories correspond to the basic RIS requirements identified in chapter 2. They are identical to the general software engineering techniques applied by MMS (see section 3.2) due to the conceptual similarity between RIS- and MMS platforms.

Separation of Concerns

- Modules, e.g., in *bolio*, *VR Juggler* (*managers*), and *MASCARET* (subsystems)
- Components, e.g., in *MR Toolkit*, *I4D*, and *VHD++*
- Nodes, e.g., in *FlowVR* (modules), *Avango*, and *SCIVE*

Execution Models

- Actors, in *SCIVE* and *Simulator X*
- Kernel, in *VR Juggler*, *NPSNET-V*, and *VHD++*
- Processes, e.g., in *DIVE*, *Avango*, *FlowVR*

Communication Schemes

- Data propagation, in *FlowVR*, *Avango*, and *SCIVE*
- Data sharing (concurrent data structures), in *MR Toolkit*, *VHD++*
- Events, e.g., in *NPSNET-V*, *VHD++*, and *Simulator X*
- Messages, e.g., in *DIVE*, *I4D*, and *FlowVR*
- Interfaces, e.g., in *MR Toolkit*, *VR Juggler*, and *VHD++* (including RMI)
- State replication, e.g., in *DIVE*, *Avango*, and *NPSNET-V*
- WebSockets, in *NPSNET-V*

State Representation Models

- Databases, in *Walkthrough*
- Entities, e.g., in *I4D* & *Unreal Engine* (actors), *REVE* (items, as part of ECS pattern), *Avango* (fieldcontainers), and *NPSNET-V* (intermixed with MVC pattern)
- HTML extensions, in *ISReal*
- Ontologies, in *ISReal* and *Simulator X*
- Semantic networks, in *SCIVE*
- UML extensions, in *MASCARET*

State Access Schemes

- Interfaces, e.g., in *Avango* and *AvangoNG*
- Semantic queries, in *ISReal* and *Simulator X*
- Semantic traverser, in *SCIVE*

In contrast to MMS publications, RIS presentations typically present and discuss the utilized execution models (11 out of 17 RIS presentations listed in Table 3.4 emphasize the execution model vs. 2 out of 17 MMS presentations listed in Table 3.3). This fosters the assessment of maintainability.

IRISs built upon RISs and (additionally) focus on a globally shared, uniform, high-level state- (R_{sta}) and behaviour (R_{beh}) representation as well as on respective explicit access schemes (R_{acc}). In contrast, MMSs rather utilize inter-submodule communication to (incrementally) process captured user input (see section 3.2). This may be the case because MMSs

oftentimes *just* conduct input analysis, while RISs have to realize applications in addition to (less complex) input processing.

Tools that support the development of RIS applications are less emphasized than amongst the presentations. Typical RIS tool support comprises compliance with external content creation applications, e.g., 3D modelling applications, monitoring tools (e.g., for *VR Juggler*), and debugging tools (e.g., for *Instantreality* and *MASCARET*), including inspectors (e.g., for *REVE* and *Simulator X*). Visual development tools (e.g., for *I4D*) are used to increase API usability at the cost of considerable development efforts. This kind of tools are thus rather available for commercial RIS, even in form of complete *Integrated Development Environment*s (IDEs) supporting visual programming (e.g., for *Unreal Engine* and *Unity*). Visual programming surely eases first steps and the creation of simple applications, however, it is not evident that its benefits for developers scale if applications become increasingly complex (Wiebusch, 2016, p. 217). In addition to tools, high level description languages (e.g., in *SCIVE* and *Simulator X*) as well as scripting languages (e.g., *Scheme* in *AVANGO*, *Python* in *AvangoNG*, and *Tcl/Tk* in *I4D*) are utilized to reduce the development effort.

Similar to MMSs, proof of concept applications are the primary method for verification in the reviewed contributions (see Table 3.4, column *Evaluation*). Performance measurements are conducted second most. In terms of maintainability, some contributions present expert reviews or lessons learned. As for MMSs, the evaluation of RIS is delicate and viable methods are rare (see section 3.1).

Discussion

With regard to software ageing, the presented RISs are slightly better available than the presented MMSs. Yet, some systems rely on nowadays uncommon operating systems, libraries, virtual machines, or hardware (e.g., *MR Toolkit*, *VR Juggler*, *NPSNET-V*, and *VHD++*), stalled third party software (*Avango* Kuck et al., 2008), are closed-source (e.g., *instantReality* and *Unity*), or are subject to strict licences (e.g., *unreal engine*). As for MMS, the overall availability is not bad. Abstracted engineering techniques, methods, algorithms, best practices and lessons learned have been accumulated and improved the field over time.

However, excessive costs for the replacement of central subsystems that are desired or required to be substituted limit the availability of older systems. This situation is attributable to close coupling caused by complex subsystem interdependencies and especially by the state representation (R_{sta}) and access (R_{acc}) requirement of RISs (Kuck et al., 2008; Latoschik & Blach, 2008; Latoschik & Tramberend, 2010). Often times this coupling is a consequence of inheritance used for state representation models—a primary method of software architec-

tures based on the object-oriented paradigm. Entity models and in particular the ECS pattern favor composition over inheritance and thus have proven to be a good solution fostering low coupling and hence increasing maintainability (Steed, 2008; Latoschik & Tramberend, 2011; Unity Technologies, 2017, and cf. Table 3.4, columns *EM* and *ECS*). Yet, these approaches possesses some deficits, especially for multimodal RIS, as discussed in the next section.

Table 3.4. Overview of selected RISs categorized into **demo**nstrations, research **(I)RIS** platforms, and **com**mercial **RIS** platforms (extended from Wiebusch, 2016, p. 35). System aspects explicitly emphasized in the reference(s) are summarized by identified issues (**Modu**larity, **Modi**fiability, **Reus(e)**ability, & **API u**sability), applied engineering techniques (**Exe**cution models, **Com**munication schemes, **Sta**te representation models, state **Acc**ess schemes, and supportive **Tool**s), and conducted evaluations (**Exp**ert reviews, **Pre**-studies, **P**roof **o**f **C**oncepts, & **Inf**ormal evaluations). The utilization of an Entity Model (**EM**) and of the Entity-Component-System (**ECS**) pattern is reported. The systems' availability for research is indicated if the source code can be obtained and if it is running on current hardware platforms (see Appendix A for details). Additional information about cells marked with * is given in the text.

Name	Type	Emphasized system aspects														EM	ECS	Available	Reference
		Issues				Techniques						Evaluation							
		Modu	Modi	Ruse	APIu	SoC	Exe	Com	Sta	Acc	Tool	Exp	Pre	PoC	Inf				
Walkthrough	demo					x			x		x			x				no	Brooks (1987)
bolio	demo					x		x	x					x				no	Zeltzer, Pieper, and Sturman (1989)
MR Toolkit	RIS	x	x		x	x	x	x	x					x				limited*	Shaw, Green, Liang, and Sun (1993)
DIVE	RIS	x	x		x	x	x	x	x					x		x		no	Carlsson and Hagsand (1993), Frécon (2004)
Avocado (Avango)	RIS				x	x	x	x	x	x						x*		limited*	Tramberend (1999)
VR Juggler	RIS	x			x	x	x	x			x			x				limited*	Bierbaum et al. (2001), Allard, Gouranton, Lecointre, Melin, and Raffin (2002)
I4D	RIS	x	x	x	x	x	x	x	x		x			x		x		no	Geiger, Paelke, Reimann, and Rosenbach (2000)
NPSNET-V	RIS	x			x	x	x	x	x							x*		limited*	Kapolka, McGregor, and Capps (2002)
VHD++	RIS	x		x	x	x	x	x	x					x	x			limited*	Ponder et al. (2003)
FlowVR	RIS			x		x	x	x										yes	Allard et al. (2004)
SCIVE	IRIS	x	x	x	x	x	x	x	x	x				x		x		no	Latoschik, Froehlich, and Wendler (2006), Fröhlich (2014)
AvangoNG	RIS	x		x	x	x	x	x	x	x		x				x*		yes	Kuck et al. (2008)
ISReal	IRIS					x		x	x	x								yes	Kapahnke, Liedtke, Nesbigall, Warwas, and Klusch (2010)
instantReality	RIS				x	x			x		x							limited*	Behr, Bockholt, and Fellner (2011)
REVE	IRIS	x		x	x				x		x			x		x	x*	yes	Anastassakis and Panayiotopoulos (2012)
MASCARET	IRIS				x				x		x			x		x		yes	Chevaillier et al. (2012)
Simulator X	IRIS	x	x	x	x	x	x	x	x	x	x	x	x†	x	x	x	x	yes	Latoschik and Tramberend (2011), Fischbach et al. (2017), † see chapter 7
Unreal Engine 4	com. RIS	—	—	—	—	—	—	—	—	—	—	—	—	—	—	x*	x*	yes*	EPIC GAMES, INC. (2017)
Unity	com. RIS	—	—	—	—	—	—	—	—	—	—	—	—	—	—	x	x	limited*	Unity Technologies (2017)

3.4 Summary

Over the years, abstracted findings like software engineering techniques, methods, algorithms, and even best practices or lessons learned have been gathered in the areas of MMSs and RISs. These are the prime achievements when it comes to repeatability and ability to build on previous results. Yet, it would be highly beneficial if any (old) system could still be used. Excessive costs for the replacement of central subsystems that are desired or required to be substituted indeed limit or even hinder the availability of older systems. Seven out of the presented MMS platforms respectively twelve out of the presented RIS platforms are availability for researches. When it comes to the combination of RIS- and MMS functionality, e.g., to realize multimodal interfaces for situated interaction environments, available solutions are rare. In combination with the shortage of suitable maintainability evaluation methods for such complex systems, there is little scope to comprehend and improve the issues. In fact, maintainability is often assessed a posteriori, by developing a system and reflecting on the actually required maintenance effort.

Chapter 4

Multimodal Real-time Interactive Systems

This chapter analysis the issues arising from the combination of RISs and MMSs and identifies promising approaches to a solution that lead to the software engineering techniques proposed by this thesis. The implementation of a multimodal interface for a situated interaction environment can be conducted by following one of two principal approaches. By utilizing an independent MMS platform in combination with a RIS platform or by utilizing a MMS that is integrated into a RIS platform. In the latter case, the underlying system will likely be an IRIS, since integration of AI methods as well as a KRL is required for multimodal processing anyway.

4.1 Independent Multimodal System Usage

Independent MMS platforms (see Table 3.3) support the access of sensors capturing multiple modalities, the processing of captured data, and its fusion (see left side of Figure 4.1). In addition, these systems provide at least a communication scheme for passing results between subsystems and a data representation model. Some system also provide a globally shared application state (**A**), holding data that is relevant for more than one subsystem, in addition to internal states that subsystems may manage. The communication of final fusion results, e.g., a user command that shall trigger an action ($Rbeh$), as well as relevant intermediate results to the RIS platform is left to the application developer. Typical solutions comprise the addition of a dedicated subsystem that forwards information, e.g., using sockets.

RIS platforms likewise provide an execution model for subsystems and at least one communication scheme (see right side of Figure 4.1). Most systems also provide a global state representation (**B**) that can be accessed by all subsystems as well as means to integrate SDKs and drivers for sensor data access.

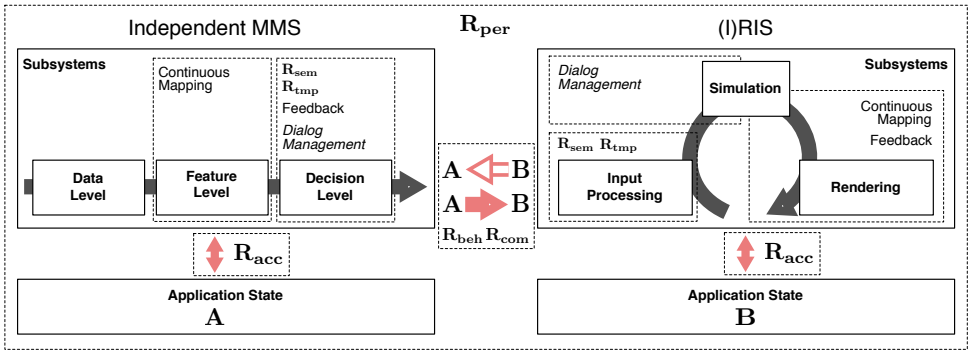

Figure 4.1. Expectable architecture an independent MMS platform utilized in combination with an (I)RIS platform. The intra-system organization is simplified with grey arrows indicating the logical flow. Solid boxes (blocks) illustrate relevant system parts. Dashed boxes highlight which key requirements have to be considered where.

Since the state representation models of the chosen MMS and RIS will most likely not be the same, the communication of (intermediate) results requires a conversion $\mathbf{A} \rightarrow \mathbf{B}$. The concrete amount of required communication—and thus conversion—depends on the features the application requires. Continous mapping requirements of user input to application state elements exacerbate these communication needs. Moreover, some features require more expressive communication protocols or alternatively the additional communication of application state changes from RIS to MMS $\mathbf{B} \rightarrow \mathbf{A}$, i.e., a state synchronization. For instance, referents indicated by an indefinite article or pronoun can be communicated as wildcards together with respective pointing directions or resolved within the MMS platform if relevant application state properties are synchronized (to satisfy R_{sem} and partly R_{tmp}).

Access and synchronization of application states is crucial in terms of maintainability and performance (red arrows in Figure 4.1). System-internal access is inevitable and less problematic. Synchronization between systems, however, is a potential source for low performance, due to the required serialization and conversion, as well as for ad-hoc implementations with poor maintainability.

Even more importantly, some features can be realized within both systems (italic in Figure 4.1), e.g., a dialog management subsystem, or require adaptions in both systems, like semantic constraint checks (R_{sem}), feedback at the time of processing, and continuous mapping (in the absence of state synchronization). This distribution of functionality serving one concern decreases coherence, adds additional complexity, and thus reduces maintainability.

4.2 Integrated Multimodal System Usage

RIS-dependent MMSs (see Table 3.3) do not suffer from these issues. Basically such a MMS, or rather such a RIS platform, comprises input analyzing subsystems devoted to multimodal input processing, in addition to other typical RIS subsystems (see Figure 4.2). Semantic- and temporal constraint checks (R_{sem} and R_{tmp}), feedback at the time of processing, and continuous mapping can be coherently implemented, since all subsystems can communicate or even access a uniform shared application state, without the need to convert representations or to synchronize between platforms. Features that could be implemented either within an independent MMS platform or within a (I)RIS platform, can also benefit from this coherence. For instance, a dialog management subsystem whose functionality is also used for a virtual agent realization. Still, actions have to be triggered as part of input analysis and subsequently be executed, e.g., by the simulation subsystems.

These combined systems (RIS-MMS), however, are especially prone to the coupling dilemma. One main reason may be that a multimodal RIS implies a combination of two already very demanding sets of functional- and performance requirements, which exacerbates the simultaneous fulfillment of other software qualities, like maintainability. Besides the contribution of this thesis, only one of the general RIS platforms was extended with explicit MMI support: *Avango* as basis of the *virtuelle Werkstatt* (see Table 3.3 and Table 3.4). The *virtuelle Werkstatt* has no running build or successor, though. Other MMS platforms that explicitly support RIS are either dedicated to a specific application (*ICONIC*) or strongly focus robot control peculiarities.

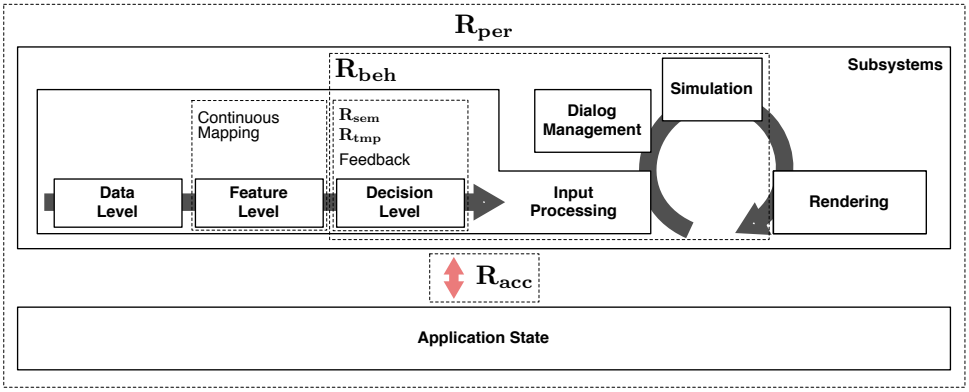

Figure 4.2. Expectable architecture of a MMS integrated into a (I)RIS platform. The intra-system organization is simplified with grey arrows indicating the logical flow of control. Solid boxes (blocks) illustrate relevant system parts. Dashed boxes highlight which key requirements have to be considered where.

4.3 Summary

The area of multimodal RIS lacks an approach that fosters maintainability *and* performance. The utilization of an independent MMS has two principle drawbacks: (1) reduced coherence for features that require functionality in both systems, (2) performance and potential maintainability issues resulting from data management, especially due to the required synchronization and conversion. This thesis thus targets multimodal input processing within an IRIS platform as well as a solution to the coupling dilemma. A uniform and subsystem agnostic state representation and access, capable of handling symbolic concepts as well as relations between those concepts, is key to a solution. There are many potential techniques, ranging from direct variable access to centralized or external storages that are accessed via a dedicated query language (Fischbach et al., 2017). Viable representatives of the former approach, e.g., graphs or hash-maps, provide high performance and the highest expressiveness, due to being defined within the RIS platform's programming langage. On the downside, these kind of approaches promote high coupling, if no further means to eliminate the reliance on specific variable or function names are applied. Representatives of the latter approach, e.g, an in-memory SQL database, strongly promote decoupling, due to their intrinsic independence from a specific RIS platform. Moreover, they facilitate semantic, i.e., symbolic, annotations as well as the structuring of content by means of realtions, e.g., a database can be extended to support Resource Description Framework (RDF) structures, e.g., by Sesame, Stardog, or Apache Jena. At the same time, however, this characteristic implies the need of converting queries into the dedicated query language, i.e., a performance overhead. In addition, modification of data with such query languages is rather limited and mostly restricted to accessing and updating contents. An ideal state representation should therefore have the benefits of the above presented contrasts and at best none of their drawbacks.

Entity models and in particular the ECS pattern are in-code methods that have proven to ease this dilemma and provide good performance (e.g., Lange, Weller, & Zachmann, 2016). Therefore they are a good foundation to build on. However, the ECS pattern does not explicitly specify (1) how to define symbols that denote entity properties and components (types) as well as symbols that represent entity properties (symbolic values), (2) how to represent relations between entities, and (3) how to make entity property- and component type information available at runtime. If not addressed, these specification gaps entail deficits that exacerbate the integration of AI methods: components from different applications (and RISs) are mostly incompatible with each other, due to a non-exportable symbol and type definition, and most importantly, access to entity properties outside a (sub-)systems's associated components is not supported, due to missing runtime type information. The latter, in par-

ticular, is a major conflict for methods that have to semantically reflect the application state and thus require to potentially access all properties of all entities. In the case of multimodal processing, the verification of semantic correctness (R_{sem}) requires this kind of access.

This thesis presents six software engineering techniques that base on the ECS pattern to benefit from its expressiveness and performance as an in-code solution and that extend the pattern by intermixing it with common independent knowledge representation methods to counter its prevalent deficits. By this means, all fundamental state representation and access requirements (R_{sta} and R_{acc}) of multimodal IRISs are satisfied without forfeiting performance (R_{per}) and maintainability. Beyond that, the approach allows to uniformly define behavior (R_{beh}) and to decouple threads of execution by semantic descriptions of their data sinks and sources in the application state. The techniques comprise the utilization of an external ontology facilitating the definition of concepts and relations as well as the automatic generation of in-code equivalents. Thus, even rendering a partial inter-RIS exchangeability possible.

The utility of the techniques is showcased by means of a reference implementation, comprising an integrated MMS, that combines them with an execution model (R_{exe}) and a communication scheme (R_{com}). Moreover, it is illustrated how three technique independent design choices of this reference implementation additionally foster maintainability. A walkthrough of central implementation aspects of the motivated use case reveals how the realization of typical multimodal processing requirements (R_{syn}, R_{tmp}, R_{sem}) benefits from the presented techniques and the taken design decisions. Finally, the six techniques as well as the reference implementation are validated by applying all main methods found in literature expert reviews, proof of concept prototypes, pre-studies, and informal evaluations.

Chapter 5

Semantics-based Software Techniques

This chapter presents the results of all software development and scientific discussion activities related to this thesis in their highest abstraction as six generalized concepts (Fischbach et al., 2017). Due to their reliance on a semantic grounding of identifiers, these concepts are referred to as semantics-based software techniques. Each technique is showcased on the basis of the interaction use case. Conclusive discussions emphasize maintainability benefits arising from an utilization of the techniques within a RIS architecture. The techniques are independent from a specific programming language and not dedicated to other architecture concepts, except the entity model. Thus other researchers and developers can benefit from this techniques by implementing one or more of them in their own system(s).

Division of Labor

Besides the mentoring by the supervisor of this thesis, the presented semantics-based software techniques as well as the reference implementation elaborated in chapter 6 have been developed with a team of three researchers: Stephan Rehfeld, Dennis Wiebusch, and Martin Fischbach (the author). General architecture concepts have principally been advanced by all team members. The author's share is emphasized by his conference and workshop publications and reflected by the selection of software parts from the reference implementation that are elaborated in the next chapter. The devision of labor is a consequence of the different focus areas that each team member pursued. Stephan Rehfeld focussed on synchronization an distribution concepts for IRISs (Rehfeld, 2017). Dennis Wiebusch focussed on the decoupling of simulation modules, application content, and application logic (Wiebusch, 2016).

5.1 Semantic Grounding

The first technique is a grounding mechanism for elementary system aspects at the core level of a system. The most basic building blocks of this technique are symbols that signify concepts that are relevant for APIs, like properties and behavior of (virtual) environment elements as well as of the system itself. They are utilized to facilitate a common ground for communication and access, e.g., within interface definitions, and are thus called *grounded symbol*s. Grounded symbols can be used directly as values of properties. In order to describe properties and parameters so called *semantic type*s are utilized. A semantic type is a tuple consisting of a grounded symbol and a data type, i.e., an assignment of meaning to a data type. A concrete value of a semantic type is called a *semantic value*. It is a triple consisting of a grounded symbol, a data type, and a value of that data type. Relations between the underlying concepts, mostly denoted by prepositions and mathematical operatiors, are modelled as *semantic functions* that are associated with semantic types and -values. Grounded symbols, semantic types, -values, and functions are meant to be defined as first class citizens of the target programming language, separately from concrete implementations as well as from concrete interfaces. Figure 5.1 illustrates the technique's integration into a typical IRIS architecture, while Figure 5.2 shows principal implementation characteristics as well as example definitions required for the interaction use case.

The *semantic grounding* technique enables the reflection of properties and parameters even if the target programming language does not support that level of self-inspection for their data types. Moreover, it allows the consideration of the meaning of properties and param-

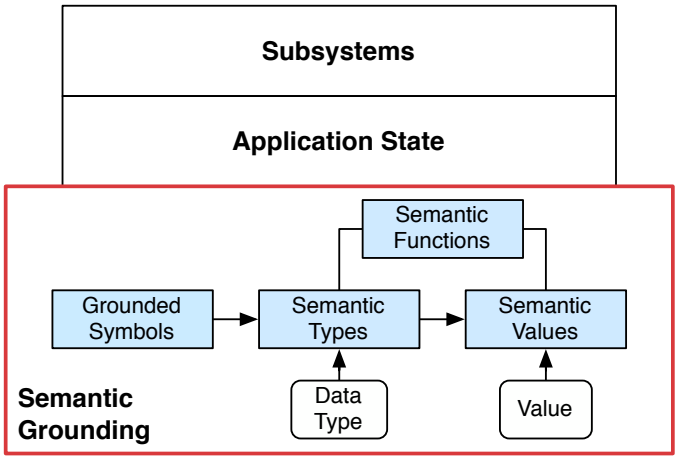

Figure 5.1. An IRIS architecture utilizing semantic grounding (red block).

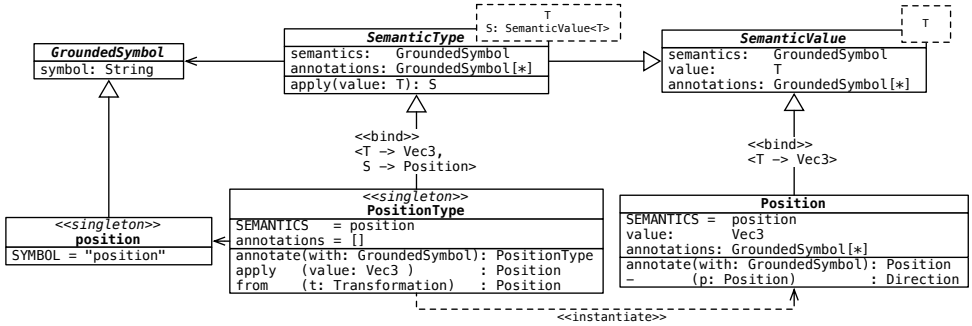

Figure 5.2. Object-oriented modeling of `GroundedSymbols`, `SemanticTypess`, and `SemanticValues`. Semantic types can be combined with a concrete value (`apply`) to create a respective semantic type. Both can be associated with one or more annotations (`annotate`) to refine their semantics. A concrete example is the representation of a `Position` in 3-dimensional space. Semantic functions are associated with their return type or with a semantic value, depending on their application in prefix- or infix notation, respectively. For instance, the `from` function of class `PositionType` (prefix) or the `-` operator of class `Position` (infix). The counterintuitive inheritance relation between `SemanticType` and its superclass `SemanticValue` is intended, since it allows concise definitions, e.g., in interfaces where a certain property is desired and its value may be undefined (cf. Wiebusch, 2016, pp. 107–109 and the definition of class `Statement` in Figure 5.6).

eters at compile time, apart from their data types. A color property, for instance, can be distinguished from a position property, even if both are represented by floating point arrays. Semantic functions are a means to describe relations between properties that allow a procedural generation. They facilitate an automatic inference of properties, e.g., if a radius is specified but a diameter is inquired, and further improve the suitability of semantic types and -values as building block for internal DSLs. Ultimately, the semantic grounding technique allows to decouple the agreement on identifiers and their meaning from concrete interface definitions, e.g., possible property names and types from the utilized state representation's API. It thus serves as a basis for state- and behavior representation techniques (R_{sta} and R_{beh}) presented in the next sections. The separate definition also facilitates the opportunity to reuse grounded symbols, semantic types, and -values in multiple, possibly unrelated applications or even RIS platforms.

5.2 Semantic Entity-Component State

The second technique refines the concept of semantic entities (Latoschik, 2005) by means of semantic grounding. As in any entity model, every object that can be distinctly identified and

that is meaningful to the system or application is represented as an entity (R_{sta}). Entities can represent virtual objects perceptible by the user, but also input devices, users, and subsystem configurations. The semantic entity-component state technique models entities as sets of semantic values. Semantic values represent entity properties, which can be added, removed, and accessed using semantic types and -values at runtime (R_{acc_1}). Entity properties include relations to other entities, i.e., semantic values with underlying data type `Entity`.

This basic concept can further be complemented by incorporating the ECS pattern's main idea to gain a structuring mechanism for system and application logic (see Wiebusch, 2016, pp. 73–86 for details). Semantic values can be grouped to correspond to the ECS pattern's *component* concept. Such components are identified by a grounded symbol. In contrast to many ECS-pattern implementations, these components only comprise descriptions of entity properties, i.e., semantic types, rather than the values itself. Special subsystems realize the ECS pattern's *system* concept and, i.a., define which components they operate on. Whenever a component is added to or removed from an entity, e.g., at its creation or during its lifetime, all associated subsystems are notified. These subsystems primarily use entity properties covered by their associated components. However, they are not restricted to and can access any property. Figure 5.3 illustrates the technique with the aid of the interaction use case.

An interface concept for accessing entity properties is shown in Figure 5.4. Entities are globally accessibel, identified by *Universally Unique IDentifier*s (`UUID`s), composed of `SemanticValues`, and additionally described by `Components`, which determine a part of the entity's properties. The type of a `Component` is used by subsystems to indicate an association. Entity properties can be set by means of `SemanticValues`. `SemanticTypes` are used to obtain the value of an entity property, either once (`get`) or whenever it is changed (`observe`). The latter access utilizes a `Callback` function that defines the reaction on a value change. A similar approach is feasible for the definition of `get`, i.e., `get(property, callback)`, to enable the interface to be used together with synchronous as well as with asynchronous execution schemes. Callback executions then either occur instantly or delayed. An supplementary synchronization layer may be added to counter race conditions, which are inherent to asynchronous global state access by multiple subsystems.

This technique utilizes separately defined first class citizens of a traget programming language to realize the access to entity properties. It thus decouples the entity model implementation from the semantics of concrete applications and from concrete system aspects, in contrast to implementations based on mutator methods or direct variable access. Due to the traits of semantic types and -values, ECS pattern deficits are countered: the externalized definition fosters inter-application reuse, while the self-inspection capabilities facilitate the reflection on and access to entity properties outside a subsystem's associated components.

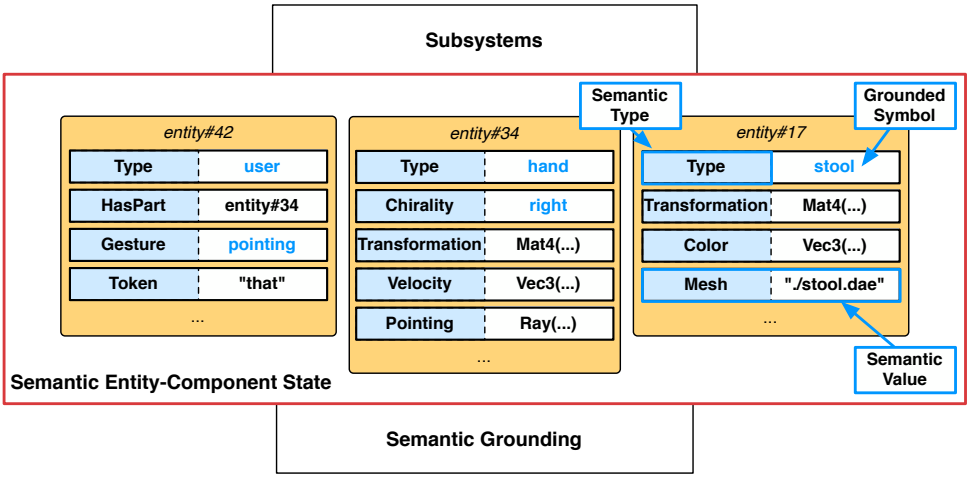

Figure 5.3. Excerpt of an exemplary global application state (red block) representing the environment during the use case. Semantic values are the building blocks of entities and represent their properties. Semantic types are used to reference these properties. Grounded symbols are used as entity property values denoting high-level symbolic concepts, like `pointing`. Relations between entities are denoted by semantic values of data type `Entity`, e.g., `HasPart(entity#34)`. Physical-environment properties captured by sensors are represented uniformly to VE properties, e.g., a tracked joint of the user (*entity#34*) or her last word spoken (`Token` property of *entity#42*).

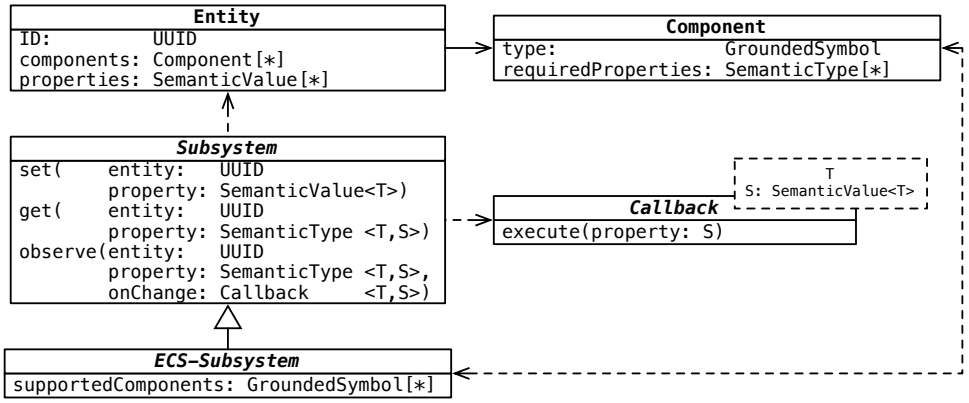

Figure 5.4. Concretization of the semantic entity-component state technique. Refer to the text for details.

5.3 Grounded Actions

The third technique utilizes semantic grounding to describe application behavior, i.e., operations that are triggerable by the user via the interface and that alter the application state in a way that can be perceived by the user (R_{beh}). Actions consist of a set of preconditions, a set of parameters, and a set of effects. Each action is associated with a grounded symbol that represents its meaning, promoting it to a *grounded action*. Parameters are defined by semantic types that act as placeholders for semantic values, which have to be passed for an execution. Preconditions and effects are defined by means of semantic values and -types, specifying properties that entities have to possess, including relations to other entities. In order to be usable, actions have to be bound to an implementation in the target programming language that procures that the preconditions are transitioned into the effects using the passed parameters.

Figure 5.5 illustrates the integration of grounded actions into an IRIS architecture and shows an example action. Inside the behavior block, a concrete grounded action is illustrated, representing the collocation of two objects in the context of the use case. It is associated with the grounded symbol *collocate*. Its invocation requires two parameters of type *Entity*, annotated with *subject* and *object*. The preconditions require that the subject entity is *Moveable* and that the target entity (object) has a *Position* property. Its effect is that the subject is near the target, represented by a *Near* property of the subject entity that denotes a corresponding relation to the object entity.

A concretization of grounded actions is sketched in Figure 5.6. They are comprised of a `GroundedSymbol` describing their semantics, an implementation `Callback`, and an `ActionDescription`. The latter specifies parameters as `SemanticTypes` and preconditions as well as effects as `Statements`. Annotations to semantic types are used to keep multiple parameters of the same semantic type apart, e.g., by annotating two parameters that each represent an entity with *subject* and *object*, respectively. Statements base on predicate logic and comprise a subject entity, a predicate, and an object. Annotations specified in the `ActionDescription`'s parameter set are used to match subject (and object) to passed semantic values. Consequently, a parameter set has to contain at least one entity to enable the implementation to yield its effects into the application state and to allow statement definitions. Four types of statements are intended: (1) `HasProperty` denoting that an entity has a certain property regardless of its value, (2) `HasValue` denoting that an entity has a certain property with a constant value, (3) `HasParameter-AsProperty` denoting that an entity has a certain property that is equal to one of the parameters (especially dedicated to define effects), and (4) `HasRelation` denoting that an entity has a certain relation to another entity. In order

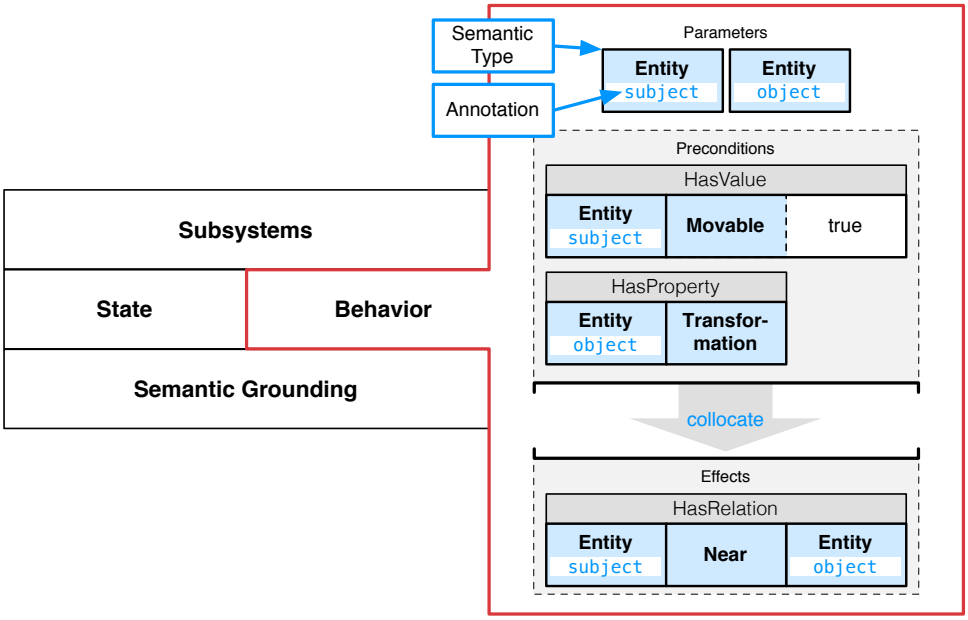

Figure 5.5. The grounded action technique (**Behavior**, red block) complements the semantic entity-component state representation (state), and builds upon the semantic grounding technique. Refer to the text for details.

to verify a statement for a given set of parameters, these four statement types are mapped to two entity property validators, called `EntityFilter`: `HasProperty`, checking if a concrete entity has a concrete property regardless of its value (directly matching 1) or `HasValue`, checking if a concrete entity has a property with a concrete value (matching 2–4). In the latter case, the concrete values are either constant for the statement (2) or have to be taken from the parameters passed to `validateFor` (3 and 4). In case of `HasRelation` (4), the statement's predicate determines the entity property to check and the entity denoted as object its value, e.g., `Near(<objectEntity>)` as showcased in Figure 5.5.

Altogether, grounded actions provide a means to semantically describe application and system capabilities in consistency with the state description. They decouple description, implementation, and invocation, which facilitates their reuse in other applications, if these applications share the required semantic grounding. Subsystems can reflect on grounded actions, due to the explicit modelling of parameters, preconditions, and effects as well as due to the semantic grounding. This reflection can be supported by the utilization of existing planning software, since action descriptions can easily be transformed into fragments of common definition languages, like the Planning Domain Definition Language (PDDL, McDermott et al., 1998). For instance, if the preconditions of a grounded action are not com-

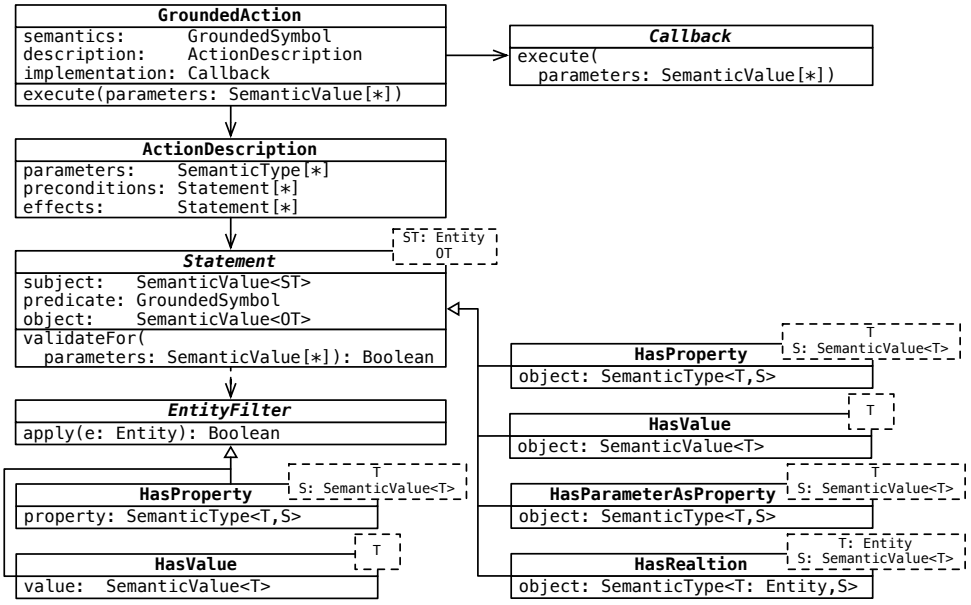

Figure 5.6. Concretization of the grounded action technique. Refer to the text for details.

pletely met, a planner can automatically infer a sequence of actions that lead to a state that permits the execution. Thus the realization of virtual agent behavior and declarative user interfaces are facilitated. Alternatively to the latter, grounded actions are also beneficial for realizing instruction-based (multimodal) interfaces. Grounded actions are typically related to concepts that denote verbs in those scenarios. During decision-level fusion, e.g., processed verbal phrases can be utilized to select a corresponding grounded action. The action's preconditions and parameters then serve as semantic constraints (R_{sem}) that have to be fulfilled by the rest of the utterance. For instance, the grounded action *collocate* could correspond to the verb *to put* and be retrieved when processing the respective token. The analyzing subsystem can trigger the action's execution, if further processing of the user's utterance yields an entity that is movable and another entity that it can be moved to.

5.4 Semantic Queries

The fourth technique allows subsystems to perform state and behavior queries by means of semantic descriptions, yielding semantic entities and grounded actions (R_{acc_2}). A specialized subsystem serves as central registry for entities and actions. It can be queried by other subsystems (as an extension of the *world interface* presented by Wiebusch, Latoschik, & Tram-

berend, 2010). Results of semantic queries shall be usable by subsystems without involving the central registry again.

There are three pragmatic methods to query a semantic entity, by means of: (e1) an associated grounded symbol, (e2) a logical combination of desired properties and property values, and (e3) a grounded action, for which entities that satisfy the preconditions are sought. Conceptually, all these three methods come down to semantic descriptions of desired properties that are formalized by utilizing semantic types and -values in from of *entity filter*s. These filters are either custom-built (e1 and e2) or obtained from grounded action preconditions (e3). Moreover, (e1) is a special case of (e2) if the grounded symbol that is associated with an entity is modeled as one of its properties, e.g., of semantic type *Type*.

Similarly, grounded actions can principally be queried by means of: (a1) an associated grounded symbol, (a2) a logical combination of desired effects, and (a3) a semantic entity, for which actions are sought whose preconditions are satisfied by the entity. (a1) comes down to a simple lookup based on the grounded action's associated grounded symbol (`semantics`). (a2) implies the use of a planning algorithm if the queried effects do not completely match one registered action's effects. In this case, the answer to a query can contain a sequence of actions, whose successive invocation leads to the desired effects, or can contain zero actions, if no such sequence exists. (a3) is complementarily to (e3), in the sense that entity filters are obtained from the preconditions of all registered actions and are applied to identify matches.

Figure 5.7 illustrates semantic queries in the context of an IRIS architecture. Figure 5.8 details the technique by showing a concretized central registry interface. The central registry subsystem holds references to all entities created by other subsystems as well as all actions registred by them. Subsystems can inquire entities from the central registry using `EntityFilters` by calling one of two methods. (1) `requestRegisteredEntities` queries all entity references for which an invocation of the passed entity filter's `apply` method (passing the referenced entity) yields `true`. (2) `observeRegisteredEntities` registers the subsystem to be notified if an entity appears in the application state, for which an invocation of the passed entity filter's `apply` method yields `true`. Subsystems can inquire actions from the central registry by calling `requestRegisteredAction` and passing a `GroundedSymbol` that identifies the action. `EntityFilter` correspond to those used for the grounded action technique (cf. Figure 5.6), though they are extended by functions that allow their logical combination to facilitate semantic query creation.

Figure 5.9 exemplifies an entity filter in the context of the interaction use case. During multimodal fusion, it can be used to retrieve the green chair the user pointed at by querying all entities that represent a chair, have a *Transformation* property, are of color green, and are near a given pointing ray. The requirement of the transformation property as well as

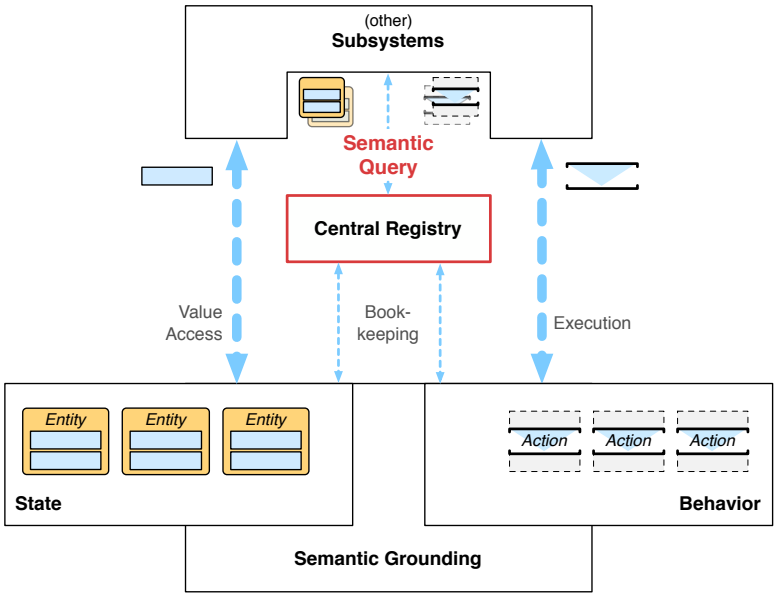

Figure 5.7. The central registry (red block) is a special subsystem that observes application state changes, especially entity creation and removal, as well as action registrations (bookkeeping). Other subsystems can request actions and entities by means of semantic queries (red text). The answer to such a query comprises references to concrete entities or actions that match the query. The typically much more frequent access to values of queried entities and the execution of queried actions does not involve the central registry any more.

of the color value is realized by using the `EntityFilters` `HasProperty` and `HasValue`, respectively. The check for the entity being a chair (*IsA*) can be realized by using a `HasValue` filter if this entity characteristic is modelled accordingly, e.g., as a property of semantic type *Type*. The check for being near a given ray uses a dedicated `EntityFilter` that implements the specific arithmetics.

The semantic query technique allows subsystems to use entities and actions without requiring explicit references that are otherwise typically passed from creating- to using subsystems. This decouples subsystems in terms of their data sinks and sources as well as of their utilization of application and system functionality. Thus, it facilitates the reuse of subsystems in other contexts or applications, as long as the required application state and behavior representation elements exist. For instance, a subsystem that operates on the transformation of the user's right hand does not need the respective entity reference nor does it need to know the subsystem that created this entity. A semantic query using an entity filter that checks if an entity has the properties *Type(hand)* and *Chirality(right)* can be used instead, e.g., upon creation of the subsystem.

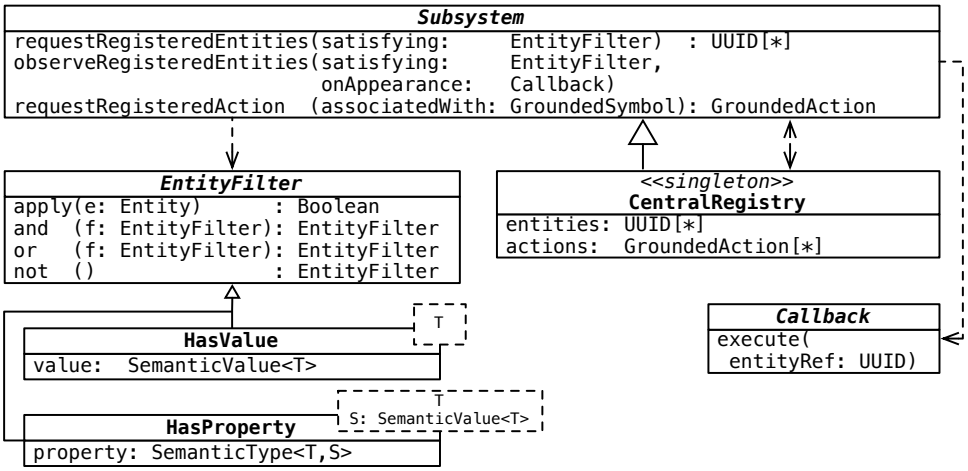

Figure 5.8. Concretization of the semantic queries technique. Refer to the text for details.

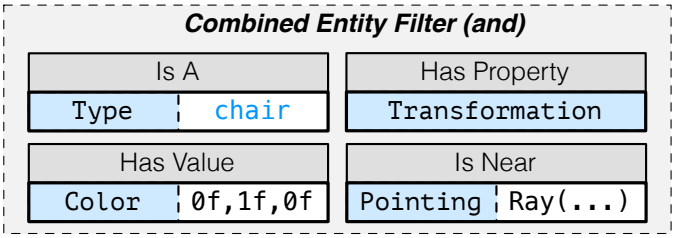

Figure 5.9. Illustration of a logically combined entity filter in the context of the interaction example.

Semantic queries facilitate entity lookup based on semantics. While (e1) and (a1) are rather straight forward, the remaining access methods provide high flexibility for subsystems. (e2) further counters the ECS pattern's deficit of unprovided access to entities outside a subsystem's component associations. Moreover, (a2) allows to retrieve grounded actions even if they have been defined differently, e.g., *move* and *collocate*, based on effect equality. This is highly beneficial for the incorporation of AI methods and especially for decision-level multimodal fusion. The uniform access to actions allows fusion implementations to retrieve potential actions based on the same elements that natural language is composed of: commonly accepted identifiers, which, in turn, can be semantically reflected due to the grounded action technique (see section 6.3 for an example application). Beyond that, (e3) and (a3) allow to interrelate state and behavior, allowing to question what can be done with a specific entity and which entities permit a specific action. The corresponding answers are likewise beneficial for user interfaces, to be able to show a user what she can do and to what she can apply commands, as well as for virtual agents that require a similar level of VE introspection.

Altogether, the technique is comparable to semantic query languages like SPARQL (W3C, 2017). However, it avoids parsing overhead due to being defined within the target programming language. In addition, entity properties can be accessed and actions executed without involving the central registry, thus avoiding a potential performance bottleneck.

5.5 Code from Semantics

The fifth technique is a generalization and abstraction layer for the semantic grounding-, the semantic entity-component state-, and the grounded action technique. Counterparts to grounded symbols, semantic types, -values, -functions, components, and grounded actions are defined in external ontology files and transformed into native code of a traget programming language. The thus generated first class citizens comprise a reference to their corresponding external concept that facilitates lookups and reasoning within the ontology at runtime. The transformation process is intended to be integrated as an automated step into the development toolchain and to be supported by appropriate tools. Figure 5.10 concretizes the external modelling of concepts as well as the automatic code generation idea: meta-concepts (red boxes) as well as grounded symbols (black boxes) are represented in the ontology as classes, concrete occurrences of semantic types, components, and grounded actions are modelled as instances (ellipses). Relations between classes and instances define the remaining properties of the first-class citizens to be generated. Relations from instances to classes (illustrated with a double filled arrow) are chosen for the sake of clarity. Within concrete ontology implementations, they may be realized with the aid of additional constructs,

like a representative instance of each `GroundedSymbol` subclass. During runtime, the generated language primitives can be unambiguously matched to their corresponding ontology entry be means of an additionally generated *Internationalized Resource Identifier* (`IRI`, red text).

The *code from semantics* technique is an independent supplement to the other techniques, since all generation results (e.g., classes) could also be defined manually. It enables the utilization of the associated ontology at compile- and runtime, e.g., to access additional concept relations or apply reasoning, due to the generated IRI for each primitive. Thus, additional facts can be inferred from existing information and be integrated into the application state or be directly used. This is beneficial for checking semantic constraints (R_{sem}) during decision-level fusion. For instance, the grounded action *collocate* can be retrieved from the central registry by querying actions associated with the verb *to put*, since this relation is modelled in the ontology (see lower left of Figure 5.10: *to_put* is an instance of class *Verb* with a relation to the *CollocateAction* instance). Later during the fusion process, the implementation of the *IsA* filter (cf. Figure 5.9) can apply a reasoning operation on the associated ontology to determine the user's intention, even if the application state does not contain an entity of type *chair* fulfilling all requirements, but rather one of type *stool* (see lower left of Figure 5.10: class *Chair* is a subclass of class *Stool*). Moreover, the ontology can also be enhanced with application state information to extend the reasoning capabilities, e.g., realized by a dedicated subsystem that synchronizes (selected) application state changes into the ontology.

In terms of maintainability, the externalized agreement on names and meaning of classes, functions, and variables that are used to define interfaces for state- and behavior representation and -access, increases cohesion and further decouples data model, action descriptions, and concrete subsystems. In addition, it facilitates communication between (I)RISs by enabling the reuse of interface definitions for other systems (see Figure 5.11). It thus overcomes a limitation of ECS-pattern implementations. This reuse is further facilitated, if the ontology is split into several coherent parts that are joined by import mechanisms. Commonly available knowledge source, e.g., an ontology of verbs, can hence be incorporated.

Altogether, the *code by semantics* technique facilitates fast access to application state and -behavior (R_{per}), due to the generation of native code, as well as the utilization of elaborated knowledge representation- and reasoning methods (R_{sta_2} and R_{sem}), due to the externalized definitions. The technique implies slightly longer building times, since the generation process has to take place before compiling. Potential development overhead, e.g., for editing the ontology, has to be countered by tool support to keep development efficient. This issue is eased if the choice of the concrete ontology implementation also considers the existence of suitable editing tools that can be used directly or extended to the development needs.

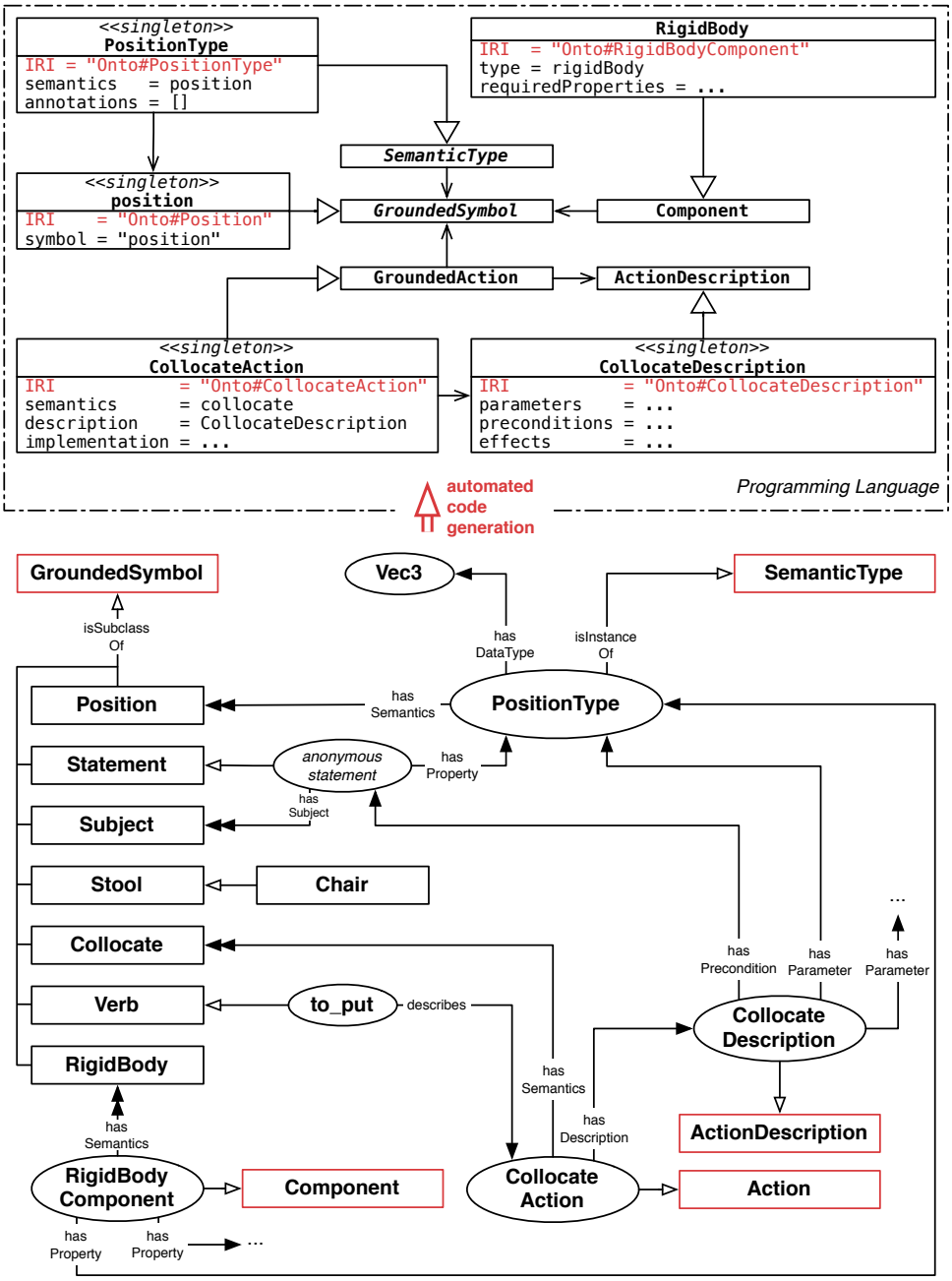

Figure 5.10. The *code from semantics* technique exemplified by means of the interaction use case: an ontology excerpt (lower part) suitable for generating grounded symbols, semantic types, components, and grounded actions as first class citizens of a target programming language (upper part). Semantic functions (not shown) are modelled similar to grounded actions, devoid of preconditions and effects. Refer to the text for details.

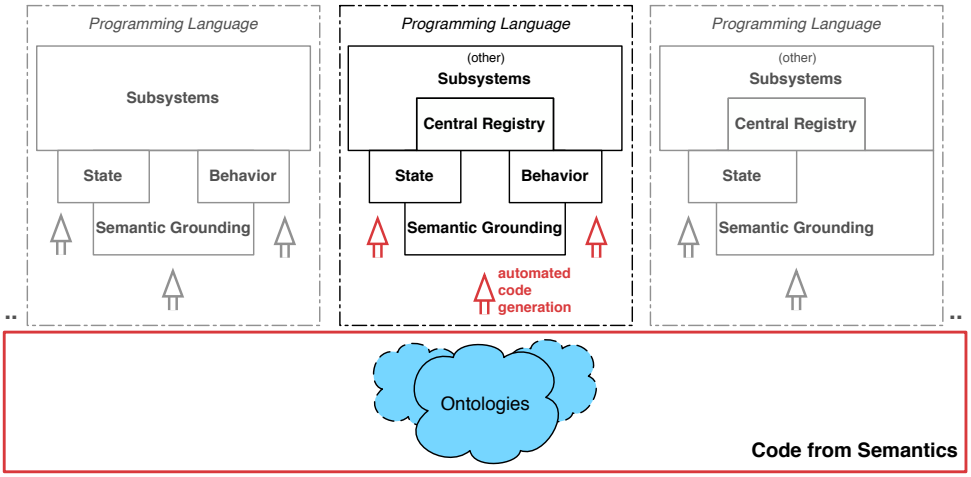

Figure 5.11. The *code from semantics* technique as basis of multiple (I)RIS architectures. Its independence from a concrete programming language facilitates the reuse of definitions.

5.6 Decoupling by Semantics

All five techniques presented so far contribute to decoupling based on a semantic description of interfaces. Technique six joins them to provide an abstraction layer for the definition of subsystems. Entities and actions, required by subsystems, are referenced by describing their characteristics, rather than by passing them as reference on creation or via callbacks. These descriptions are intermixed with calculation rules that are to be evaluated in reaction to state changes. The abstraction layer includes a high-level API for defining *processing steps*, while encapsulating the necessary management, e.g., execution of semantic queries and observation of entity properties. Figure 5.12 illustrates the technique by three exemplary multimodal processing steps in the context of the interaction use case.

At data-level, a processing step may require the transformation of the (user's) right hand (Data-level, line 4) to extract its position. *Transformation* denotes a matrix-based representation that includes position information and that is typically used to describe kinematic chains, like the human skeleton. The entity representing the right hand is identified by two characteristics, being of type *hand* and of chirality *right* (Data-level, lines 1–2), which correspond to entity filters that can be combined to create a semantic query. On creation a hence described subsystem, shall use this entity filter to obtain a concrete entity reference and to observe this entity's transformation property. Whenever this property changes, the subsystem shall evaluate the statement in curly braces and update the entity's properties with the result, i.e., the position extracted by means of a semantic function (Data-level, lines 5–7).

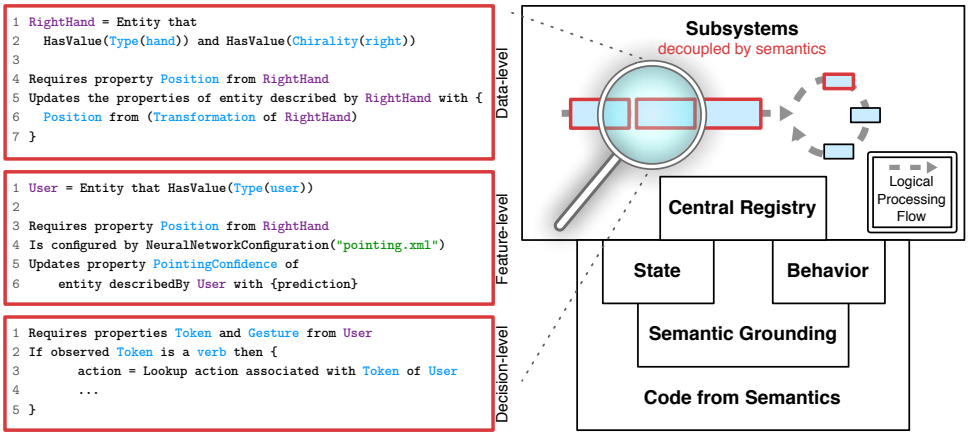

Figure 5.12. The decoupling by semantics techniques applied to describe requirements and effects of subsystems (right, red boxes). Some subsystems may still be associated with entities by means of the ECS-pattern's component mechanism (black boxes). Human-readable, high-level description sketches of three multimodal processing steps at data-, feature-, and decision-level (left). The descriptions utilize grounded symbols, semantic types, and -values (blue). They follow Sire and Chatty's (2004) proposal of ideal languages features and comprise variables (purple text) and control structures. Refer to the text for details.

At feature-level, a processing step may embed a more complex evaluation based on a neutral network that analyzes the trajectory of the right hand to predict a pointing gesture. Additional description elements are used to reference an externalized configuration (Feature-level, line 4) required for such a specialized processing step. However, data sources (Feature-level, line 3) and sinks (Feature-level, lines 5–6) as well as reactions to changes (Feature-level, line 6) are described using the same elements as before.

At decision-level, a processing step may fuse speech and gesture and thus requires to observe spoken tokens and performed gestures (Decision-level, line 1). The description constitutes an excerpt of a typically more complex specification that connects application state changes to registred grounded actions (Decision-level, lines 2–5). Spoken tokens that are identified to be verbs can be used to inquiry associated actions at the central registry (Decision-level, line 3). The description of a thus obtained action can be used as semantic constraint, i.e., the rest of the utterance has to fill the defined parameters and to satisfy the preconditions. If so, its execution can be occasioned. An example using this lookup is given in section 6.3.

Altogether, the decoupling by semantics technique is an approach to jointly utilize all previously presented techniques by means of high-level API that foster usability for developers. It allows to define interrelations between subsystems and the application state as well as between subsystems and the application behaviour based on semantic descriptions. On the one

hand, it adds an alternative to the ECS-pattern's component mechanism for decoupling subsystems and state that is more flexible, especially in terms of granularity. On the other hand, it adds a novel mechanism to decouple behavior in a similar fashion. Due to the application of semantic queries, subsystems can be reused in other contexts as long as the required application state and behavior representation elements exist.

5.7 Summary

The six presented software techniques extend the ECS-pattern's basic principle and provide solutions to its runtime type deficit as well as its incompatibility issues between RISs. More importantly, the techniques improve component granularity, facilitate access to entity properties outside a subsystems component association, and additionally provide means to semantically describe behavior as complement to the state representation. Ultimately, the six techniques constitute solutions to the fundamental (I)RIS requirements (see Table 5.1). They facilitate decoupling by means of semantic interface descriptions beyond common alternatives, while providing additional benefits, like improved state access flexibility, a consistently designed behavior representation, compatibility with common reasoning and planning approaches, the support of high-level APIs, and inter-RIS compatibility. They thus foster several central aspects of maintainability while being especially suitable for AI methods. The guiding use case constitutes an instruction-based, intentional MMI. The six techniques, however, are not limited to this scenario. Access to a state- and behavior representation based on semantic descriptions as well as planning and reasoning capabilities may likewise facilitate the analysis of unintentional non-verbal communication and the generation of multimodal output.

The presented techniques are independent of a concrete programming language. Developers can benefit by integrating them in their own software or use them to enhance third-party systems. Their dependencies are listed in Table 5.1, in case a subset of them is required.

Table 5.1. Dependencies between the presented techniques and their contributions to solve the fundamental (I)RIS requirements identified in chapter 2. Each row represents a technique and indicates which other techniques are required (r) or beneficial (b) for its implementation (Revised from Fischbach, Wiebusch, & Latoschik, 2017).

		1	**2**	**3**	**4**	**5**	Contributes to
Semantic Grounding	**1**						R_{acc}
Semantic Entity-Component State	**2**	r				b	R_{sta}
Grounded Actions	**3**	r				b	R_{beh}, R_{sem}
Semantic Queries	**4**	r	b	b		b	R_{acc}
Code from Semantics	**5**	r					R_{per}, R_{acc}
Decoupling by Semantics	**6**	r	b	b	r	b	R_{exe}

Chapter 6

Reference Implementation

This chapter elaborates on a reference implementation of the semantics-based software techniques presented in chapter 5. All respective development efforts have contributed to the current version of an IRIS platform called *Simulator X* (Latoschik & Tramberend, 2011; Fischbach et al., 2017), which is available under an open-source licence Wiebusch, Fischbach, Rehfeld, Tramberend, and Latoschik, 2016. It is dedicated to system architecture research in the area of VR and MR with a special focus on multimodal interfaces. Consequently, the platfom comprises an integrated multimodal input processing framework, called *miPro*. The presentation of the author's share of the implementation efforts is divided into the following aspects: design decisions, core implementation, multimodal input processing, and ancillary contributions. Taken architectural and practical design decisions that are not specified by the techniques are justified first. The implementation of each technique is detailed thereafter, jointly constituting the core of the Simulator X platform. The utilization of this implementation for multimodal input processing is showcaed within the context of the interaction use case. Finally, ancillary contributions are presented, including additional subsystems, application development abstractions, as well as code examples that supplement the documentation to support beginning developers.

6.1 Design Decisions

Some decisions that are essential for an implementation are left open by the presented semantics-based software techniques: the choice of an execution model, of a communication scheme, of an ontology, and of a programming language. In the case of Simulator X, the former two are ensured by applying the actor model, making it three taken decisions that are introduced and justified in this section (cf. Wiebusch, Fischbach, Latoschik, & Tramberend, 2012).

Actor Model

The actor model (Hewitt, Bishop, & Steiger, 1973) defines *actors* as independent flows of control that communicate solely via messages. It thus constitutes a solution to the fundamental execution and communication requirements of RISs (R_{exe} and R_{com}) that uses actors as basic building blocks realizing concurrency. The actor model follows the paradigm "*everything is an actor*". Once created (by another actor), an actor exclusively can respond to a message received by one of the following three theoretical actions:

1. send message
2. create further actors
3. change its reaction to future messages

All computations required in the context of an IRIS have to be performed within these theoretical constraints. In practice, however, callbacks from input device drivers are oftentimes additional triggers for sending messages as well as rendering is oftentimes a further reaction to a message received. Besides obvious applications of messaging, e.g., for passing results, this mechanism can also be used to realize instruction sequences that have to be repeated within a certain time window, e.g., rendering an image every 10 ms. To do so, an actor can send itself a message upon reaching the end of such a loop. Actor model implementations may support this implementation by providing a means to schedule the dispatch of a message.

The actor model is chosen, since it is a scalable execution model with an inherent support for communication, thus facilitating distribution and event systems. The former is highly beneficial for large-scale (I)RIS applications, e.g., VR, the latter for decoupling the execution of subsystems (Steed, 2008). Its scalability mainly results from the paradigm "*everything is an actor*". Actors can one-to-one realize the common coarse grained separation of computational aspects into subsystems, e.g., input processing, physical simulation, and 3D rendering. In addition, the model is especially designed to enable the concurrent use of actors for fine grained and occasionally necessary computations. One the one hand, this accommodates the typical behavior of AI methods, where analysis, reasoning, or planning, has to be performed sporadically with required processing times that are hard to exactly predict. On the other hand, this flexibility allows to fully exploit available hardware capabilities, ranging from multiple processing cores over CPUs to cluster nodes. Finally, the limitation to message passing for communication—and thus for data sharing—avoids the necessity of common concurrency control mechanisms, such as mutexes. Synchronization mechanisms may still be required and added at higher layers.

Despite these benefits, some specific features of the actor model have to be considered upon its utilization for a RIS. The actor model makes no assumptions about message ordering. Actors run concurrently and message passing is asynchronous, thus this is obvious for most of the communication. Yet, two messages sent in a specific order from one actor to the same receiver could also overtake each other. One reason can be that the two actors are running on two different machines and are communicating via a network protocol that does not guarantee ordering itself. However, such incidents must be excluded to enable correct implementation of access and communication mechanisms building on message passing between actors (cf. Wiebusch, 2016, pp. 119–126). Most actor model implementations provide message ordering between actors and guarantee that multiple messages from one actor to the same receiver are processed in the order they are sent, e.g., by realizing the actor's mailbox as first-in-first-out data structure. In networked scenarios additional means are necessary, e.g., relying on a suitable network protocol, like TCP, or numbering messages.

Furthermore, the debugging and profiling of actor systems beyond local computations is not straightforward. Common supportive tools are often ineffective. For instance, stack traces by debuggers end in a thread managed by the actor model implementation's scheduler without helping the developer to identify where a certain message went or came from. Violated restrictions related to implementation particularities of the actor model lead to malfunctioning that is hard to identify. Messages have to be immutable to not violate the principle that actors exclusively communicate via messages. A nevertheless passed mutable data structure may create a situation in which two actors mutually access a shared memory unguarded, i.e., a race condition. The resulting malfunctioning is typically difficult if not impossible to reproduce. Here, appropriate abstraction layers should hide implementation particularities of lower layers if possible, avoid incorrect accidental misuse in the first place, and thus increase API usability. The concrete abstraction layers of the Simulator X platform are presented in section 6.2. In addition, dedicated tool support not only assists developers that are inexperienced with the actor model, but also further increases API usability as well as analysability in general (cf. Rehfeld, Tramberend, & Latoschik, 2014).

Finally, the actor model does not specify the structure or meaning of messages. Thus additional agreements are necessary to guarantee that an receiving actor *understands* a message, i.e., is able to properly process it. The semantic grounding technique specifically fulfills this requirement and complements the actor model in this regard.

Web Ontology Language

The Web Ontology Language (OWL, W3C OWL Working Group, 2009) is a description language for ontologies developed to serve as a long-term standard for representing knowledge in the world wide web. OWL is formally based on description logics and is consequently capable of representing concepts (classes), individuals and their realtions. Due to its web dedication, all resources are referenceable by IRIs. It thus fulfills the basic requirements posed by the code from semantic technique. Moreover, its applicability for describing state and behavior within an IRIS has been shown by the *ISReal* framework (Kapahnke et al., 2010).

In addition to that, the choice of OWL is driven by further benefits. OWL ontologies can be partitioned into multiple files linked by import statements and IRIs, fostering modularity and reusability. Sharing and collaboration are essential for OWL's designated use in the context of the world wide web as they are when it comes to the fostering of reusability for IRISs. The broad dissemination of OWL can, on the one hand, be exploited by incorporating existing ontologies into applications (e.g., the *Wordnet* ontology by Miller, 1995, to facilitate reasoning about verbs and their mapping to actions during multimodal fusion). On the other hand, existing tools can be potentially readily used or be extended to better match development use cases. These tools include, graphical OWL editors, like Protégé (Musen, 2015) and the Fluent Editor (Cognitum, 2015), as well as reasoning software, like HermiT (Glimm, Horrocks, Motik, Stoilos, & Wang, 2014) and Pellet (Sirin, Parsia, Grau, Kalyanpur, & Katz, 2007). Finally, OWL promotes the RIS performance requirement (R_{per}) by supporting sublanguages, called OWL profiles, that trade expressive power for reasoning efficiency. OWL profiles do not have to be explicitly specified, but rather result form the language elements used. This performance benefit can be further increased by utilizing ontology partitioning. Thus reasoning can be more easily restricted to a certain relevant subset of the complete ontology in use, which typically results in a performance gain. Refer to Wiebusch (2016, pp. 47–48 and 55–98) for a comprehensive justification of the OWL choice.

Scala

Scala (Odersky et al., 2004) is a general purpose programming language that provides object-oriented as well as functional paradigms. Its source code is intended to be compiled to bytecode for the *Java Virtual Machine* (JVM). Consequently, Scala can be easily intermixed with Java libraries. At the time the decision for an programming language taken, a similar Scala compiler yielding bytecode for *Microsoft*'s *.Net* virtual machine was available. Its support, however, is discontinued by now.

The choice for Scala bases on the following benefits of the language with respect to (I)RIS

```
1  object Simple {
2    def main(): Unit = {
3      val ping     = () => {println("ping")}
4      val pong     = () => {println("pong")}
5      val pingPong =
6        Chain function ping andFunction pong
7      pingPong() //Executes the chained function
8    }
9  }
10
11 object Chain {
12   def function(f: () => Unit) = {
13     new SequenceOfFunctions(List(f))
14   }
15 }
16
17 class SequenceOfFunctions(functions: List[() => Unit]) {
18   def andFunction(f: () => Unit) = {
19     new SequenceOfFunctions(f :: functions)
20   }
21   def apply() = functions.reverse.foreach(_.apply())
22 }
```

Listing 1. Illustration of Scala's syntax flexibility and support for internal DSLs. Keywords are depicted in dark blue and local variables in purple. Singleton-like patterns can be defined using the keyword `object` alternatively to `class`. This way an exemplary main function is defined (lines 2–8). A function is assigned to the variables `ping` and `pong` respectively (lines 3 and 4). For that, two lambda expressions are used, each defining a function that takes no parameters and prints to the console. Its return type `Unit` is analogous to Java's `void`. The type of these variables is not specified, it is implicitly determined by the type of the statement right of the = operator by Scala's type inference mechanism. The result of an internal DSL statement, i.e., the concatenation of `ping` and `pong`, is assigned to the variable `pingPong` (lines 5–6) and executed afterwards (line 7). The DSL statement makes use of the option to omit the dot operator or the parentheses in certain cases. Its implementation is shown below (lines 11-22). Object `Chain` defines a static function `function` that takes a further function, like `ping` (lines 12–14). This constitues the implementation of first part of the DSL statement and can be alternatively called by `Chain.function(ping)`. The result is a new instance of class `SequenceOfFunctions` which composites sequences of functions (lines 17–22). `SequenceOfFunctions` defines a member function that allows to prepend to the sequence (lines 18–20, using the `::` operator of `List`) as well as a member function `apply` that represents the `()` operator. This operator is called in line 7. Its implementation reverses the sequence order to respect the DSL statement and then executes all stored functions sequentially. Altogether, the statement in line 6 resembles a description in natural language, however, it is still restricted to the Scala syntax, e.g., preventing the separation of `andFunction` into `and function`.

development. The provided object-orient paradigm as well as Scala's close relation to the widespread Java programming language ease first steps for developers. The interoperability with Java makes a large amount of potentially useful libraries available for development. The compatibility with the JVM implicates platform independency. Furthermore, Scala comes with a profound support for the actor model. Formerly included in the Scala standard library, it is now provided through the *akka* (Lightbend Inc., 2016) library, a widespread implementation of this model. Finally, Scala's syntax flexibility (see Listing 1) greatly facilitates the maintenance of concise code as well as the creation of internal DSLs (see section 6.2).

This benefits come at one central drawback that has been traded off against Scala's gains: reduced performance. Programming languages like *C++* are assumed to provide higher performance and are for that reason even wider spread in the area of RIS development. The main cause is the manual memory management that these languages permit, enabling developers to optimize time critical and frequent access. In contrast, the garbage collection meachnims of the JVM may occupy hardware ressources in critical situations leading to compromised user experience (Stauffert, Niebling, & Latoschik, 2016), since it is not dedicated to RIS requirements. However, manual memory management capabilities imply that memory has to be manually allocated, potentially accessed, and freed using pointer arithmetics. A typical source for high development effort. Scala, by contrast, hides this management from the developer. The endeavour to provide a solution to the coupling dilemma and to increase maintainability in general primarily implies definitions of abstraction layers and interfaces that may need to be rapidly prototyped. Both of these requirements called for Scala. In addition, early proof of concept prototypes, which have been extensively continued (see chapter 7), validated the feasibility of the choice. The presented semantics-based software techniques can still be implemented in C++. The principally higher effort can be minimized by utilizing the advantage of having a clear definition now.

6.2 Core

The actor model, the programming language Scala and the web ontology language are the practical basis for Simulator X platform. The result of their utilization for implementing the semantics-based software techniques constitutes its core functionality. This section emphasizes the key mechanisms applied to implement the platform's core. The presentation focusses on interfaces as the primary means for decoupling and API usability.

Semantic Grounding

The semantic grounding concept presented in section 5.1 is implemented utilizing the following key mechanisms:

- Scala's `object` syntax
- Currying
- Implicit parameters

An implementation excerpt of the semantic value `Position` as well as of its associated grounded symbol and semantic type is shown in Listing 2. Grounded symbols (lines 1–5) are singleton wrappers for identifiers, which are stored as member variable `symbol` (line 3). They are named after the contained identifier, beginning with a lower case letter. Just as semantic types and -values, they correspond to definitions in an associated ontology, from which they are automatically generated. The link to the ontology is reflected by a member variable holding an IRI (lines 4,9, and 18).

The implementation of semantic types and associated semantic values exploits Scala's syntax for defining singleton-like patterns, called *companion objects* or *companion modules*, along-

```
1  object GroundedSymbols { /** auto-generated **/
2    object position extends GroundedSymbol(
3      symbol       = "Position",
4      ontologyLink = new IRI("GroundedSymbols.owl#Position"))
5  /*...*/}
6  object SemanticTypes { /** auto-generated **/
7    object Position extends SemanticType[Vec3,Position](
8      semantic     = GroundedSymbols.Position
9      ontologyLink = new IRI("SemanticTypes.owl#PositionType"))
10   {
11     def apply(value: Vec3)                  : Position = {/*...*/}
12     def apply(value: Vec3, timestamp: Long): Position = {/*...*/}
13     def from (t: Transformation)(implicit functions: SemanticFunctions): Position =
14       functions.positionFrom(t)
15   }
16   class Position(value: Vec3, timestamp: Long) extends SemanticValue[Vec3](
17     semantics    = GroundedSymbols.Position
18     ontologyLink = new IRI("SemanticTypes.owl#PositionType"))
19   {
20     def -(p: Position)(implicit functions: SemanticFunctions): Direction =
21       functions.-(this,p)
22 }/*...*/}
```

Listing 2. Implementation excerpt of the semantic grounding technique within Simulator X, reduced to its essentials. Concrete grounded symbols, semantic types, and -values are depicted in light blue. Refer to the text for details.

```
1  object SemanticValue {
2    def example() {
3      val properties     = new SemanticValueSet()
4      val transformation = Transformation(Mat4.Identity)
5      properties.set(transformation)
6      println(properties.get(Transformation).value)
7    }
8  /*...*/}
9  class SemanticValueSet() {
10   def set(property: SemanticValue[_]): Unit                  = {/*...*/}
11   def get[SV, ST <: SemanticType[_,SV]](description: ST): SV = {/*...*/}
12 /*...*/}
```

Listing 3. Example utilization of semantic values. Generic types are depicted in cyan.

side with a class. Thus, a semantic type may be named identical to its associated semantic value, e.g., the object `Position` realizing a semantic type (lines 7–15) and its associated semantic value `Position` (lines 16–22). As defined by the concept, the semantic type allows the instantiation of respective semantic values (lines 11–12). All semantic values contain a timestamp in addition to the raw value, to enable subsystems to check temporal constraints (R_{tmp}) and to perform other time-based processing, like interpolation. If no explicit timestamp is passed for the instantiation of an semantic value (cf. line 11), the implementation uses the current time as default value. General functions, e.g., for annotation or value access, are defined in the super classes `SemanticType` and `SemanticValue`. Functions relating to other semantic values, i.e., semantic functions, are defined in-situ (lines 13–14 and 20–21). Semantic functions are curried: in addition to a set of parameters representing relation partners, i.e., one or more semantic values, these functions take an additional set of parameters consisting of one item (`functions`). After passing the first parameter set, a semantic function hence results in another function that takes a second parameter set. The `functions` parameter realizes a reference to compatible implementations, which are called in the respective implementations (lines 14 and 21). The `implicit` keyword facilitates a concise syntax when using semantic functions.

The utilization of semantic types, -values, and -functions is showcased in Listing 3 with the aid of an related class for representing collections of properties, called `SemanticValueSet`. An exemplary utilization is shown in lines 2–7. Semantic values are instantiated by passing a concrete value to the `apply` method of the associated semantic type (line 4). Such values can be added to `SemanticValueSets` (line 5) and retrieved later on (line 6), while preserving the semantic value type. In the context of the example, this means that `properties.get(Transformation)` returns an object of type `Transformation`. `SemanticValueSet` is implemented by composition of a data structure that maps semantic

```
object SemanticValue {
  def example() { /*...*/
    properties.set(Position from transformation)
    properties.set(properties.get(Position) - properties.get(Position))
  }
  implicit object Implementations extends SemanticFunctions with Defaults {
    def positionFrom(t: Transformation) = {/*...*/}
  }
/*...*/}

abstract class SemanticFunctions { /** auto-generated **/
  def -(minuend: Position, subtrahend: Position): Position
  def positionFrom(t: Transformation): Position
/*...*/}
trait Defaults {
  def -(minuend: Position, subtrahend: Position) = {/*...*/}
/*...*/}
```

Listing 4. Continuation of the exemplary use of semantic values in Listing 3, demonstrating the application of semantic functions.

types, i.e., property descriptions, to semantic values, i.e., concrete property instances (lines 9–12). Its key features are a method to store semantic values (`set`, line 10) and to retrieve them based on semantic descriptions if existent (`get`, line 11). The signature of the get method reflects the association between semantic types and semantic values: the concrete semantic type passed to the method (`ST`) determines the return type, that is, the associated semantic value (`SV`). Semantic value sets are a possible implementation of entities, i.e., collections of properties constituting objects relevant to the simulation.

Listing 4 continues this example and demonstrates the utilization of semantic functions (lines 2–5). The `from` function is used to extract the position from the previously defined transformation, which is stored in the `SemanticValueSet` (line 3). The application of `from` is defined by making use of the option to omit dot operators and parathesis. The statement in line 3 only defines the first parameter set of the curried function `from` (cf. Listing 2). The second parameter set, consisting of one parameter of type `SemanticFunction`, is automatically resolved due to its annotation with the `implicit` keyword. If implicit parameters are not (explicitly) passed, the current scope is searched to fill in missing values. Candidates are objects of compatible type that are also annotated with `implicit`. In this case the `object Implementations` (lines 6–8). It is compatible, since it inherits from the abstract class `SemanticFunctions` (lines 10–13). `SemanticFunctions` defines all functions required for semantic types as well as -values and is likewise generated from an associated ontology. Concrete implementations may be defined with regard to reuse (`DefaultImplementations` in lines 14–15) and mixed in (keyword `with` in line 6) or defined locally (line 7). The original

statement `Position from transformation` can thus keep its concise shape while allowing to bind different implementations. The second application of a semantic function (line 4) demonstrates the infix utilization of the - operator. At the same time, the preservation of types after accessing a `SemanticValueSet` is shown, since - is a function of the concrete semantic value `Position`.

Altogether, the key mechanisms applied for the implementation result in the following benefits. The use of Scala's singleton pattern syntax results in a concise notation for instantiating and accessing semantic values. Currying in combination with implicit parameters allows to separate the implementation of a semantic function from its definition, which facilitates the automatic generation from an ontology and the binding of alternative realizations. Moreover, implicit parameters together with an `implicit` definition of semantic function implementations in super classes further support conciseness for semantic function applications.

Semantic Entity-Component State

The implementation of the semantic entity-component state concept presented in section 5.2 provides globally accessible variants of semantic value sets, i.e., entities as a collections of properties. The implementation builds upon the following key mechanisms:

- State variables
- Implicit parameters
- Traits

At first sight, the concept of globally accessible state contradicts the actor model principles, which disallow direct access to other actor contexts. The *state variable* mechanism solves this issue by providing the illusion of globally accessible variables based on message passing (see Listing 5). All subsystems are implemented as actors, including the application logic specification. An exemplary actor that operates on a passed `StateVariable` is shown in lines 1–6. This state variable mimics variable of type `Vec3` that can be potentially passed to and accessed by all actors. Setting a value is identical to common mutator method implementations (line 4). The getter method, however, takes a callback function as parameter, instead of returning the state variable's value (line 3–5). Its signature is owed to the underlying implementation. The state variable's value may be not directly accessible by the executing actor, i.e., it may not be within its context. Hence, message passing is required to instead inquire it from the actor that manages the value. The passed callback function is then executed delayed.

The key aspects of Simulator X's state variable implementation are shown in lines 7–19. A `StateVariable` itself is solely a composition of an unique identifier and a reference to an actor that manages the access to the actual state of the variable, called *owner* (line 7). Besides setting (line 8) and getting (lines 9–10) the value of a state variable, actors can register for a notification on changes to the variable's value (`observe`, line 11) and deregister from this notification (`ignore`, line 12). These methods are part of the `StateVariable` class to model a common variable interface. Concrete message passing, however, has to originate from an actor. The implementations of the methods in class `StateVariable`, consequently forward all parameters to such an actor (lines 8 and 10, `observe` and `ignore` is implemented accordingly). `StateVariableHandling` (lines 14–19) implements the required communication. It inherits from `SimulatorXActor` that provides general means to register handlers for messages based on their types, including support for matching request messages to their associated answer. The invocation of `get` and `set` shown in lines 3 and 4 exploits `implicit` parameters to improve conciseness. `StateVariableHandling` defines a constant `implicit` variable (`context`) that holds an reference to itself (line 15). Thus, to enable the use of state variables inside actors, simply the `StateVariableHandling` trait has to be mixed in as part of the actor's definition (line 1). Invocations of `StateVariable` member functions then do not require the `context` parameter to be specified, since it is automatically inferred to be `context` defined in `StateVariableHandling`.

```
1  class ApplicationActor extends SimulatorXActor with StateVariableHandling {
2    def offset(position: StateVariable[Vec3], delta: Vec3) {
3      position.get{ value =>
4        position.set(value + delta)
5    }}
6  /*...*/}
7  class StateVariable[T](val id: UUID, val owner: StateVariableActor.Reference)  {
8    def set(value: T)(implicit context: StateVariableActor) = context.set(this, value)
9    def get(callback: (T) => Unit)(implicit context: StateVariableActor) =
10     context.get(this, callback)
11   def observe(callback: (T) => Unit)(implicit context: StateVariableActor) = {/*...*/}
12   def ignore()(implicit context: StateVariableActor) = {/*...*/}
13 /*...*/}
14 trait StateVariableHandling extends SimulatorXActor {
15   protected implicit val context : this.type = this
16   def set[T](sVar: StateVariable[T], value: T) = {/* Send message to owner */}
17   def get[T](sVar: StateVariable[T], callback: (T) => Unit) = {
18     /* Request value from owner and store callback for when the answer is received */}
19 /*...*/}
```

Listing 5. The interface to the state variable implementation, reduced to its essentials. The underlying message passing strategy is elaborated by Latoschik and Tramberend (2011) and Wiebusch (2016, pp. 168–170). Refer to the text for details.

```
1  class ApplicationActor extends SimulatorXActor with EntityHandling {
2    def setPointingRay(hand: Entity) {
3      hand.get(Transformation){ t =>
4        hand.set(Pointing(new Ray(t.origin, t.zAxis)))
5    }}
6  /*...*/}
7  class Entity(val id: UUID, val owner: StateVariableActor.Reference) extends
8    StateVariable[Map[SemanticType[_,_],StateVariable[_]]](id, owner) {
9    def set(property: SemanticValue[_])
10     (implicit context: EntityHandling) = context.set(this, property)
11   def get[SV, ST <: SemanticType[_,SV](description: ST)(callback: (SV) => Unit)
12     (implicit context: EntityHandling) = context.get(this, description, callback)
13   def observe[SV, ST <: SemanticType[_,SV](description: ST)(callback: (SV) => Unit)
14     (implicit context: EntityHandling) = {/*...*/}
15   def ignore(property: SemanticType[_,_])
16     (implicit context: EntityHandling) = {/*...*/}
17 }
18 trait EntityHandling extends StateVariableHandling {
19   implicit val context : this.type = this
20   def set(e: Entity, property: SemanticValue[_]) =
21     {/* get entity value (map), add property if new, set property value */}
22   def get[SV, ST <: SemanticType[_,SV](e: Entity, description: ST, cb: (SV) => Unit) =
23     {/* get entity value (map), lookup property, perform get on property */}
24 /*...*/}
```

Listing 6. The interface to the entity model implementation, reduced to its essentials. Refer to the text for details.

The implementation of the entity model combines the semantic grounding implementation with the state variable mechanism. Listing 6 demonstrates the key aspects of this combination. Management of entity properties is achieved by means of semantic types and -values. The underlying state variables, however, are hidden, making entities the sole interface to their properties (lines 2–5). Entities support mutator methods for their properties similar to state variables, including `observe` and `ignore`. Their signature thus includes an additional parameter of type `SemanticType`, specifying which property is addressed. For example, the transformation property of an entity can thus by requested by passing the semantic type `Transformation` as well as an appertaining callback function (lines 3–5). Within the callback, the obtained transformation value `t` can be used. In this case, to add a new property to the entity (respectively to update the property if it already existed) by invoking the `set` method with a semantic value of type `Pointing` (line 4).

The `Entity` class itself is implemented as a special `StateVariable`, holding a data structure that maps property descriptions to concrete property instances (lines 7–8), similar to `SemanticValueSet`. Thus it can also be passed to and accessed by potentially all actors, without violating the actor principles. The definition of the `Entity` class' mutator methods

(lines 9–17) constitutes a combination of the respective signatures from `SemanticValueSet` and `StateVariable`. The `set` method (lines 9–10) takes a semantic value, i.e., a property to be added to the entity or to update it. The `get` method (lines 11–12) takes a semantic type, i.e., a description of an entity property, as well as a function to be called when the value of the described property is available. Similar to the signature of `SemanticValueSet.get`, the concretely passed semantic type (`ST`) determines the single parameter of the callback function—its associated semantic value (`SV`). The `observe` method (lines 13–14) and the `ignore` method (lines 15–16) enable to register or respectively deregister from a notification upon value changes of a specified entity property. They are implemented in analogy to `get` and `set`. All four mutator methods take an additional `implicit` parameter of type `EntityHandling` to forward the invocation to an appropriate actor. Concrete message passing or rather state variable access is defined within the `EntityHandling` trait (lines 18–24). Since entities are realized as state variables, their value, i.e., the mapping beween property descriptions and state variables, can be get, set, and observed. The implementation of the `set` method (lines 20–21) accesses this map and adds an entry mapping to a new state variable if required or updates the respective existing state variable. The implementation of the `get` method (lines 22–22) also accesses this map to retrieve the state variable that holds the value of the property described by the passed semantic type. The `get` method of this state variable is subsequently invoked by passing the callbeck `cb`. The methods `observe` and `ignore` (not shown) are implemented accordingly. In analogy to the `StateVariableHandling` shown in Listing 5, `EntityHandling` defines an `implicit` reference to itself (line 19), to facilitate concise definitions inside inheriting actors when using entity mutator methods. That is, the `implicit` parameter `context` can be omitted.

This basic entity model implementation is complemented with an ECS pattern implementation as proposed by the semantic entity-component state technique. It provides an entity lifecycle mechanism based on entity descriptions that are composed of components. Components themselves describe aspects of entities and determine systems (special subsystems) that realize associated functionality. Such subsystems are implemented as specialized `SimulatorXActors` that report to a central registry upon creation and can thus be notified if relevant entities are created. A *rigid body* component, for instance, may require an entity to possess (semantic value) properties, like mass, position, and velocity, and an associated physics simulation subsystem to be registered. The complemented implementation also provides additional means to reflect upon entities, e.g., check if an entity possesses a certain property. Thus an additional condition `if(hand.has(Transformation))` could be prepended to line 3 in Listing 6. A detailed elaboration of Simulator X' ECS pattern implementation is presented in Wiebusch (2016, pp. 126–138 and pp. 158–161).

Altogether, the key mechanisms applied for the implementation result in the following benefits. State variables provide the illusion of globally accessible variables without violating the actor principles. Variable access is managed by the actor that holds the actual state variable value (*owner*). Additional synchronization mechanisms, e.g., to set multiple state variables in an atomic operation, are intended to be added in higher software layers. Traits allow to conveniently equip actors with a cohesive set of communication capabilities, e.g., required to utilize state variables or entities. Implicit parameters complement the `implicit` definition of self-references in `EntityHandling` to foster a concise invocation of entity mutator methods. The underlying state variables are transparent to property access, making entities, semantic types, and -values the primary interface to the application state. As a result, the meaning of properties is directly accessible for reflection purposes, since it is represented by first class citizens linking to an associated ontology, i.e., grounded symbols, semantic types, and -values. This is an improvement over the common practice of encoding the meaning of values in variable names, like *position* in lines 2–4 of Listing 5.

Grounded Actions

Technique three is implemented based on grounded symbols, semantic values, and -types. Planning support is realized by the use of the PDDL and the library *planning4j* (AMIS group, 2017). A comprehensive elaboration is given by Wiebusch (2016, pp. 91–92 and pp. 174–175).

Listing 7 illustrates the interface for describing (lines 1–7) and registering (line 11) actions. The processing of the interaction use case necessitates that two objects are collocated to each other at some point, e.g., the green chair and the table. The `ActionDescription` describes this action by specifying a grounded symbol that represents its meaning (line 2), a set of preconditions (lines 3–5), and a set of effects (line 6). Preconditions and effects are specified with the aid of an internal DSL. Each line describes a concrete instance of the `Statement` concept proposed in Figure 5.6. The action's parameters are implicitly defined by the entities and semantic values that are referenced within the preconditions and effects. In this case, the action has two parameters of type `Entity` one annotated with the grounded symbol `subject` and one with the grounded symbol `target`. The preconditions require that the subject to possess a property `Moveable` with value `true` (line 3) as well as a property `Transformation` of arbitrary value (line 4) and the target to also possess a property `Transformation` of arbitrary value (line 5). The effects are that there is a relation `Near` between subject and target, i.e., both entities posses a property `Near` whose value is a reference to the other entity, respectively. An implementation of this description is indicated in line 9. Its compatibility to the description cannot be checked automatically. It is thus the developers responsibility to

```
1  object Collocate extends ActionDescription(
2    identifier      = collocate,
3    preConditions   = Set(Entity named subject hasProperty Moveable(true),
4                          Entity named subject hasProperty Transformation,
5                          Entity named target  hasProperty Transformation),
6    effects         = Set(Entity named subject is Near the Entity named target)
7  )
8
9  def collocateImplementation(parameters: SemanticValue[_]*){/*...*/}
10
11 registerAction(Collocate, collocateImplementation)
12
13 def collocateEntities(subj: Entity, tgt: Entity){
14   Planner.accomplish(subj is Near the tgt) }
```

Listing 7. Action description (lines 1–7), implementation (indicated in line 9), registration (line 11), and execution (using a planning subsystem in line 13–14) of a grounded *collocate* action that moves one entity (the `subject`) near an other (the `target`, revised from Fischbach, Wiebusch, & Latoschik, 2017). The action's description, the access to the `parameters` collection, as well as the action's execution are based on grounded symbols and semantic types. Refer to the text for details.

ensure that the implementation really transfers the preconditions to the effects. After being recorded at the central registry (line 11), the action can be used, e.g., with the aid of a planning subsystem as shown in lines 13–14. Instead of calling an action directly, a desired state is declared. In this case, the declaration matches the effects of the `collocate` action. Hence, the planning algorithm may check the action's preconditions. If satisfied, the `collocate` action can be executed. Otherwise, a sequence of other action leading to a satisfaction is inferred and executed before, if existent.

Semantic Queries

The implementation of the semantic query concept presented in section 5.4 allows actors to request entities and actions from a central registry. This registry (*world interface* Wiebusch, 2016, pp. 161–162) is a special kind of actor known to all other actors. It is notified upon entity as well as action creation and stores respective references. The implementation of the semantic query concept builds upon the following key mechanisms:

- Context Binding
- Delimited Continuations
- Traits
- Singleton Actors

```
1 abstract class EntityFilter extends Function[Entity,Boolean@CPSRet]{
2   implicit var context: EntityHandling = null
3   def rebindContext(newContext: EntityHandling) =
4     { context = newContext }
5   def and(that: EntityFilter) = new EntityFilter {/*...*/}
6 /*...*/}
```

Listing 8. The base class for entity filters, reduced to its essentials. Refer to the text for details.

Requests to the central registry use the entity filter to describe desired entities. Its implementation is shown in Listing 8. In principle, an entity filter is a function that takes an entity and returns a boolean value, which specifies if an entity matches the filter (line 1). However, a potential access violation has to be considered, since entity filters are functions that will likely access entity properties and at the same time are meant to be communicated between actors, e.g., from a requesting actor to the central registry. Entity mutators utilize the surrounding actor context when invoked (cf. section 6.2). If an entity filter that applies a get operation, for instance, is sent to the registry and invoked there without further precautions, it will still use the requesting actor's context. This most likely leads to malfunction of both actors. To counteract this issue, the entity filter implementation defines an own `implicit` context variable (line 2). If an entity mutator is used in an implementation of `EntityFilter`'s `apply` method (inherited from `Function[Entity,Boolean@CPSRet]`), it uses the innermost matching `implicit` variable, i.e., the one define in line 2. To allow the registry to set a received entity filter's context to itself, `EntityFilter` defines a rebind method (lines 3–4). In addition to this fundamental functionality, entity filters can be logically combined, e.g., by means of the `and` operation defined in line 5.

Concrete entity filters have to to implement the `apply` method. If an entity property is to be obtained with such an implementation, then using the `get` method of class `Entity` (cf. Listing 6) is not a viable option, since its callback mechanism does not allow to directly return a boolean value. Entity filter implementations thus make use of an alternative `get` variant based on delimited continuations (Wiebusch, 2016, pp. 150–152). This use is shown in Listing 9 by the `HasValue` implementation. `HasValue` is one of the basic entity filters proposed in section 5.3 and section 5.4. It checks if an entity possesses a specified property and if this property equals a specified value. Consequently, its constructor takes a semantic value for matching and extends `EntityFilter` (line 1). The implementation of `apply` first checks if the entity possesses the property and then retrieves its value using the alternative `get` variant (line 2–3). When using the normal `get` variant, the callback that is to be invoked when the answer to the request is available is defined explicitly. With the alternative variant it is defined implicitly by means of the *shift* and *reset* operators of Scala's

```
1 class HasValue[T](v: SemanticValue[T]) extends EntityFilter {
2   override def apply(e: Entity) =
3     { e.has(v.semanticType) && Result.of(e.get(v.semanticType)) == v }
4 }
```

Listing 9. A concrete entity filter that checks if an entity possesses a specified property and if it equals a specified value.

delimited continuations mechanism. *Reset* marks a slice of code that is reified into a function. *Shift* is used inside the *reset* block to mark sections that become parameters to that function. In the case of `HasValue`'s `apply` method *reset* could mark the complete line 3. *Shift* is used within `Result.of()` and marks the `get` request. Effectively, this leads an inversion of control, where `e.get(v.semanticType)` is executed first. Its results, or rather a received answer message, is then passed to the reified function. In this case, `result => e.has(v.semanticType) && result == v`. Similar to the *shift* operation being hidden within `Result.of()`, *reset* is hidden in `SimulatorXActor`, marking any message handler completely. One consequence of this integration of delimited continuations into the general message handling is that return types calculated by means of *shift* operations, have to be specially annotated, like `Boolean@CPSRet` (line 1 of Listing 8).

An exemplary application of entity filters for semantic queries is shown in lines 1–7 of Listing 10. Therein, an entity representing the user's right hand is selected from the application state and passed to a callback function that may process this entity further (line 2–6). It is assumed that the application state is modeled as proposed in Figure 5.3. The entity filter `rightHandFilter` is defined as a conjunction of two `HasValue` filters, checking if an entity possesses a property `Type` of value `hand` and a property `Chirality` of value `right` respectively (line 3). This filter is applied together with the `callback` function to query the central registry (lines 4–5). `getRegisteredEntities` is inherited from the `SemanticQueries` trait (lines 8–13), which equips `ApplicationActor` with communication capabilities for semantic queries in the same way as the `EntityHandling` trait does for state access. Besides the `getRegisteredEntities` request (lines 9–10), this trait provides means to register for a notification on the appearance of matching entities in the application state in the future (`observeRegisteredEntities`, lines 11–12). The implementation of these two requests basically consists of sending the passed entity filters to the central registry actor and handling the answer accordingly. However, a reference to this actor is required to do so.

To allow any actor to query the central registry without passing its reference on creation, the singleton actor mechanism is applied (lines 14–19). A `SingletonActor` wraps a `SimulatorXActor` that is passed in form of a constructor function (line 14). Its prime interface is

```
1  class ApplicationActor extends SimulatorXActor with SemanticQueryHandling {
2    def getRightHand(callback: (Entity) => Unit) {
3      val rightHandFilter = HasValue(Type(hand)) and HasValue(Chirality(right))
4      getRegisteredEntities(rightHandFilter)( entities => {
5        callback(entities.head)
6    })}
7  /*...*/}
8  trait SemanticQueries extends EntityHandling {
9    def getRegisteredEntities    (f : EntityFilter)(cb: Set[Entity] => Unit) =
10     {/* query Registry singleton actor*/}
11   def observeRegisteredEntities(f : EntityFilter)(cb: Set[Entity] => Unit) =
12     {/* query Registry singleton actor*/}
13 /*...*/}
14 abstract class SingletonActor[T <: SimulatorXActor](constructor: => T){
15   def !(message : Any) = { self ! message }
16   def self : SimulatorXActor.Reference =
17     {/* Create instance if non-existent, return reference */}
18   private var reference : Option[SimulatorXActor.Reference] = None
19 /*...*/}
20 class  Registry extends SimulatorXActor {/*...*/}
21 object Registry extends SingletonActor(new Registry)
```

Listing 10. An exemplary semantic query (lines 1–7) as well as the interface to the central registry reduced to its essentials (lines 8–21). Refer to the text for details.

the standard operator for sending messages (the `!` operator, line 15), which is implemented to send a passed message to the single instance of the wrapped actor. Consequently, it creates an instance of the wrapped actor and stores a reference to it on first access and returns the stored reference on this first and all subsequent accesses (lines 16–18). The central registry actor `Registry` (line 20) is wrapped this way (line 21) and can thus be addressed by potentially any actor without its reference being directly passed.

`EntityFilters`, the `SemanticQueries` trait, and the `Registry` actor provide means to request entities based on an associated grounded symbol (1), a logical combination of desired properties and property values (2), and grounded actions (3). The first two request types are both the direct use of entity filters as showcased in Listing 10. More precisely, the first can be realized by the application of filters like `val handFilter = HasValue(Type(hand))` if the application state is modeled accordingly. The third request type also utilizes the `SemanticQueries` trait and the `Registry` actor, however, entity filters are retrieved from action descriptions. Similarly, actions can be queried by means of grounded symbols or by specifying a desired application state as well as by utilizing a planning algorithm (as described by Wiebusch, 2016, pp. 175–179). An additional variant of inquiry by means of symbols is showcased in Listing 11. Verbs and other parts of speech are listed within a dictionary and mapped to semantic values and grounded actions. By using this dictionary, which is meant

```
object Dictionary {
  val verbs      = Map("put"   -> Collocate(), /*...*/)
  val adjectives = Map("green" -> Green(),     /*...*/)
  val nouns      = Map("chair" -> Chair(),     /*...*/)}
object Chair() extends Noun      { val entityReference    = Type(chair) }
object Green() extends Adjective { val associatedProperty = Color(Constants.green)) }
object Collocate() extends Verb  { val actionReference    = collocate }
```

Listing 11. Mapping of parts of speech to semantic values and grounded actions (Zimmerer, Fischbach, & Latoschik, 2016): verbs denote actions, adjectives entity properties, and nouns entities.

to be automatically generated from the associated ontology, the central registry can also be queried by string tokens, e.g., obtained from a subsystem wrapping an ASR. This translation from natural language elements to entities and actions is especially beneficial for decision-level multimodal fusion incorporation natural language, as showcased in section 6.3.

Altogether, the key mechanisms applied for the implementation of semantic queries result in the following benefits. Context binding provides the means for communicating functions that access the application state between actors and thus enable the implementation of entity filters. Delimited continuations facilitate the implementation of concrete filters by allowing the filter's `apply` method signature to be kept straight forward, with no additional callback functions involved. Their use in the general application state mutator interfaces was refrained from, since it necessitates the annotation of return types (e.g., `@CPSRet`) if results of such a mutator application shall be passed to other functions or be stored in local data types. In the case of entity filters, this annotation is transparent below the `EntityFilter` base class. In general use, this transparency could not be completely maintained, which was experienced to be a greater source of errors than of improvement. Traits are utilized to equip actors with communication capabilities necessary to query the central registry, which can be addressed without reference passing due to the singleton actor mechanism.

Code from Semantics

The implementation of the code from semantics technique presented in section 5.5 deals with OWL modeling and the actual generation of Scala code from OWL. It utilizes the following key mechanisms:

- Tool Support
- Process Integration

```
<!-- GroundedSymbols.owl -->
<Declaration>
  <Class IRI="#Position"/>
</Declaration>
<SubClassOf>
  <Class IRI="#Position"/>
  <Class IRI="Core.owl#GroundedSymbol"/>
</SubClassOf>
```

Listing 12. OWL definition of the grounded symbol *Position* (adapted from Fischbach et al., 2017).

```
<!-- SemanticTypes.owl -->
<Declaration>
  <NamedIndividual  IRI="#PositionType"/>
</Declaration>
<ClassAssertion>
  <Class            IRI="#SemanticType"/>
  <NamedIndividual  IRI="#PositionType"/>
</ClassAssertion>
<ClassAssertion>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#hasSemantics"/>
    <Class          IRI="GroundedSymbols.owl#Position"/>
  </ObjectSomeValuesFrom>
  <NamedIndividual  IRI="#PositionType"/>
</ClassAssertion>
<ObjectPropertyAssertion>
  <ObjectProperty   IRI="Core#hasDataType"/>
  <NamedIndividual  IRI="#PositionType"/>
  <NamedIndividual  IRI="DataTypes#math.Vec3"/>
</ObjectPropertyAssertion>
```

Listing 13. OWL definition of the semantic type *PositionType* (adapted from Fischbach et al., 2017).

Grounded symbols, semantic types, components, and grounded actions are modeled in OWL as described by Wiebusch (2016, pp. 162–165). An example modeling of a grounded symbol and a semantic type, based on the structure proposed in Figure 5.10, is shown in Listing 12 and Listing 13, respectively. The grounded symbol *Position* is an OWL class (lines 2–4) that is a subclass of the *GroundedSymbol* class (lines 5–8). The semantic type *PositionType* is an OWL individual (lines 2–4) that is a member of class *SemanticType* (lines 5–8). *PositionType* also belongs to an anonymous class that describes individuals which have the semantics *Position* (lines 9–15). This class assertion realizes the relation between individuals and classes (proposed in Figure 5.10) by means of OWL's `ObjectSomeValuesFrom` expression. The `ObjectProperty` *hasSemantics* can thus be used like a predicate denoting the

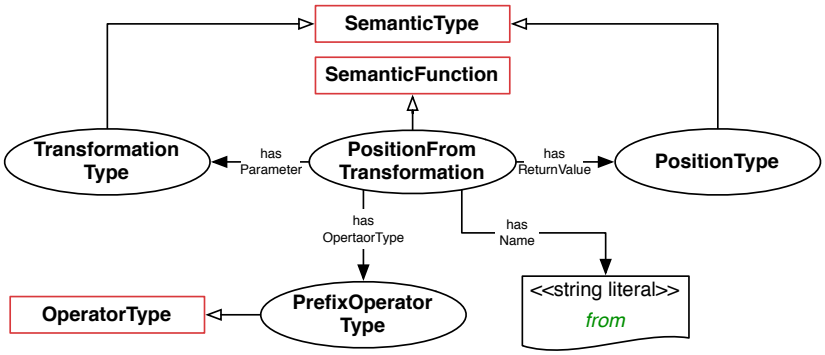

Figure 6.1. Modeling of a semantic function that represents an operation extracting the position information from a transformation matrix.

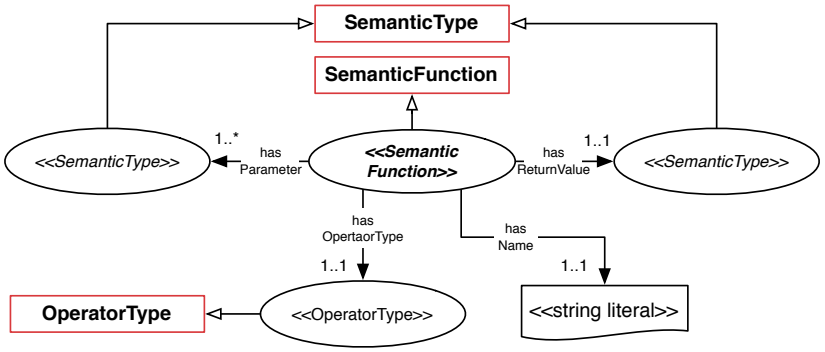

Figure 6.2. Blueprint for semantic function modeling in OWL. Refer to the text for details.

relation between the individual *PositionType* and the class *Position*. Finally, *PositionType* has a relation to another individual that represents a concrete Scala data type (lines 16–20).

Semantic functions build upon those definitions and describe relations between properties that allow a procedural generation (cf. section 5.1). They are modeled in OWL similar to grounded actions, mainly differing in the fact that they possess no preconditions and effects. Figure 6.1 shows an exemplary modeling of the semantic function *PositionFromTransformation*. The function is an individual of class *SemanticFunction* with relations to *semantic type* individuals that represent its parameter *TransformationType* and its return value *PositionType*. It possesses an additional relation to *PrefixOperatorType*, an individual of class *OperatorType* specifying that the function is used in prefix notation, as well as to a *string literal* data property that specifies the function's name.

A general blueprint for semantic functions is illustrated in Figure 6.2. Semantic function individuals have one or more parameters, exactly one return value, and exactly one operator type. Those properties are specified by means of the OWL object properties *hasParameter*,

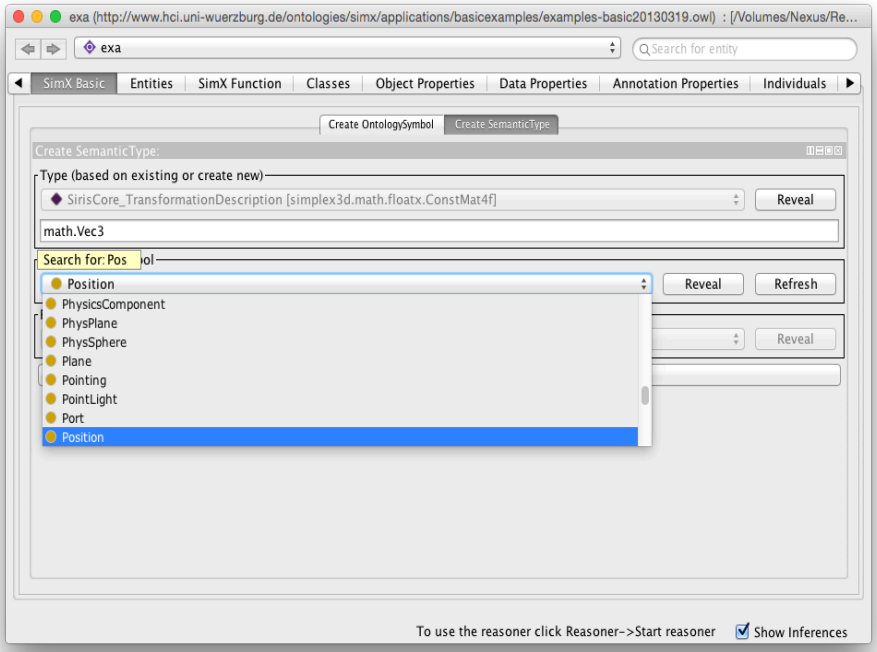

Figure 6.3. The OWL editor *protégé* with the dedicated plugin, showing the view for specifying a semantic type. Data types can be entered directly or be selected from existing semantic types. In the latter case, the user can select from a drop-down menu. Associated semantics, i.e., grounded symbols, can also be selected via drop-down menus. Both menus can be searched by typing while the menu is open.

hasReturnValue, and *hasOperatorType*. The function's one name is specified by means of the OWL data property *hasName*, which has to be of type *string*.

Jointly, the definitions specified in OWL form the basis for the RIS implementation in Scala. OWL editing thus is part of the development process implied by the proposed techniques. However, the developer is not meant to edit raw OWL files. Existing OWL tools ease the modeling effort considerably. OWL ontologies associated to Simulator X applications are best edited with *protégé* (Musen, 2015) using a dedicated plugin. This plugin extends *protégé* with views for specifying grounded symbols, semantic types, and -functions (see Figure 6.3). It is implemented in Java according to *protégé*'s plugin mechanism using the *OWL API* library (Horridge & Bechhofer, 2011) as well as the *JIDE* library for Java Swing (JIDE Software, 2017).

All grounded symbols, semantic types, and -functions, components, and grounded actions modeled in OWL are used to generate Scala first class citizens, including most code snippets shown beforehand in section 6.2. The software piece that performs this generation task is written in Scala, also using the the *OWL API* library. It is integrated into the development process by means of the build tool that is used to compile and run Simulator X applications in general (*sbt* Lightbend Inc., 2017). Thus the generation task is guaranteeing that all concepts defined in OWL are truly available to the compiler.

Altogether, the application of tool support and process integration results in an improvement of developer usability, particularly, in a reduction of development effort. Ignoring the necessity of these mechanisms, e.g., intending the associated ontology to be edited with plain *protégé* or even with a text editor, can easily lead to a situation where the development costs and complexity prevent a utilization of the associated ontology at all. The dedicated protege plugin reduces effort by mapping concept blueprints to *protégé* views, like the one presented for semantic functions. The complexity of arbitrary OWL editing is thus hidden and erroneous definitions are prevented. Moreover, the implementation of the *protégé* views constitutes an implicit formal definition of the blueprints, besides diagrams, just as the implementation of the code generation task. However, the usefulness of languages dedicated to explicitly describe OWL structures should be researched in future work.

Decoupling by Semantics

The decoupling by semantics technique presented in section 5.6 is implemented utilizing the following key mechanisms:

- Traits
- Semantic Entity References
- A Domain Specific Language
- Implicit parameters
- Local Histories
- Interpolation

The proposed processing steps are realized by actors that are equipped with specialized functionality by means of the `Processor` trait, analogous to previously presented implementations. The basic feature of `Processors` is the semantic entity reference mechanism that allows to describe required entities semantically, in contrast to passing them on creation. Listing 14 shows the relevant implementation excerpt with the aid of an exemplary

```
1  val RightHand = Type(hand) and Chirality(right)
2  class FeatureExtractor extends SemanticEntityReferences {
3    require(Transformation,RightHand)
4  }
5  trait SemanticEntityReferences extends SimulatorXActor with SemanticQueries {
6    type  SemanticEntityReference = SemanticValueSet
7    class StateUpdate[T](e:  SemanticEntityReference, newVal: SemanticValue[T]){/*...*/}
8    var   requirements: List[SemanticEntityReference, SemanticTypeSet]       = {/*...*/}
9    var   localData:    Map [SemanticEntityReference, SemanticValueSet]      = {/*...*/}
10   def require(property: SemanticType[_,_], e: SemanticEntityReference)     = {/*...*/}
11   def onRequirementUpdate(update: StateUpdate[_])
12   override def startUp()= {/*...*/
13     requirements.foreach( requirement => {
14       observeRegisteredEntities(requirement.reference.toFilter){ entities =>
15         val storage = new SemanticValueSet()
16         localData   = localData.updated(requirement.reference, storage)
17         requirement.properties.foreach{ property =>
18           entities.head.observe(property){ value =>
19             storage.set(value)
20             onNewValueOfRequirement(requirement.reference, value)
21 }}}})}/*...*/}
```

Listing 14. The implementation of semantic entity references, reduced to its essentials. Refer to the text for details.

application (lines 1–4). Essentially, a semantic entity reference is a semantic query that is meant to identify one single entity. It is defined as a conjunction of `HasValue` entity filters (cf. Listing 10) with the actual implementation of the filter being hidden from the interface. What is left, is a set of semantic values, i.e., a `SemanticValueSet`, that an entity has to possess. Line 1 shows a definition of such a `SemanticValueSet` with the aid of the convenience function `and`. An actor utilizing the `Transformation` property of the entity described by this semantic entity reference is defined in lines 2–4. The included `require` invocation (line 3) is executed during the construction of a new `FeatureExtractor` instance, since it is outside any method definition (Scala's practice to define constructors). The respective implementation is shown in lines 5–21. The `SemanticEntityReferences` trait is a `SimulatorXActor` that mixes in `SemanticQueries` to be able to request entities from the registry (line 5). Therein, required entity properties are observed automatically and their latest values are stored locally to facilitate direct access during processing. The representation of semantic entity references as `SemanticValueSets` is reflected by the type definition in line 6. Requirements are a tuple of a `SemanticEntityReference`, denoting an entity, and a `SemanticType`, denoting a property. They are stored locally (line 8) upon the invocation of `require` (line 10). The entity properties they denote are stored in a second local data structure that holds a `SemanticValueSet`, containing the latest values of required prop-

```
1  val RightHand = Type(hand) and Chirality(right)
2  class FeatureExtractor extends Processor {
3    Requires property Transformation from entity describedBy RightHand
4    Updates the properties of entity describedBy RightHand 'with' {
5      Position from (Transformation of RightHand)
6  }}
7  trait Processor extends SimulatorXActor with SemanticEntityReferences {/*...*/
8    object Requires {
9      def property(requiredProperty: SemanticType[_,_]) =
10       { /* Updates the requirements list */ }
11   }
12   object Updates {
13     /* uses onNewValueOfRequirement to register an appropriate callback */ }
14   protected implicit val context : this.type = this
15   def of[SV, ST <: SemanticType[_,SV]](property: ST, e: SemanticEntityReference): ST =
16     { /* Accesses localData and returns the respective value */ }
17 /*...*/}
18 abstract class SemanticType[T, ST <: SemanticValue[T]] {/*...*/
19   def of(entity: SemanticEntityReference)(implicit context: Processor) =
20     context.of(this, entity)
21 }
```

Listing 15. The implementation of processing steps, reduced to its essentials. Refer to the text for details.

erties, for all required entities (line 9). Upon startup of the `SemanticEntityReferences` actor by the underlying actor system, i.e., after construction, the necessary observe invocations are performed (lines 12–20). The central registry is queried for each entity referenced in the requirements (lines 13–14). Whenever an entity matching a requirement appears in the application state, including already existing ones, the local data structure is extended with a `SemanticValueSet` dedicated to store the latest values of its required properties (lines 15–16). The actual acquisition of these values is realized by means of the `observe` mechanism of entities (lines 17–20). The conflict beween the `observeRegisteredEntities` signature, potentially providing multiple matching entities, and the assumption of semantic entity references, meant to identify one single entity, is here solved by using the first matching entity (`entities.head`, line 18). The complete implementation provides at least a warning and can be configured to throw an exception, if multiple entities have matched. This part of the implementation is omitted for the sake of clarity. Whenever a required property changes, its new value is stored in the respective `SemanticValueSet` (line 19) and a callback is triggered (line 20). This callback (line 11, also cf. line 7) has to be implemented by inheriting actors, like the actual `Processor` trait shown in Listing 15.

Following the decoupling by semantic technique's proposal, the `Processor` trait provides an internal domain specific language that facilitates the definition of requirements as well as

the definition of processing rules. An example is shown in lines 2–6, using the same semantic entity reference as before (line 1). The `FeatureExtractor` requires the `Transformation` property from an entity described by the semantic entity reference `RightHand` (line 3). It updates the properties of this entity with the `Position` extracted from the required `Transformation` (lines 4–5). The processing rule (line 5) is defined using the semantic function `from` as well as an access mechanism to the local storage of the latest required properties (`Transformation of RightHand`). It is executed whenever a new value of a required property is sent to the actor. All statements of lines 3–5 are native Scala code, implemented with the aid of the Scala syntax mechanisms presented in section 6.1, yet they are nearly as readable as their natural language descriptions before. The relevant excerpt of the underlying implementation is shown in lines 7–21. `Processor` inherits from `SemanticEntityReferences` in order to use semantic entity references. The implementation of the requirement statement (lines 8–11) wraps the invocation of `require`, while the implementation of the update statement (lines 12–13) stores a callback that is executed using `onRequirementUpdate`, both shown in Listing 14. The access mechanism to the local storage is again implemented by means of implicit parameters. The `SemanticValue` base class (lines 18–21) is extended with the method `of` (line 19–20), which is invoked in line 5 to access the local storage. Its implementation uses an implicit parameter to pass itself and the semantic entity reference to the surrounding `Processor` (`context`, line 20), which accesses the local storage and returns the respective value. In total, the utilization of the `Processor` trait (lines 2–6) shows an example where one property is required. If multiple parameters are required, the processing rule may not be invoked when the first value arrives, but only after a value is available for each requirement. Such considerations are taken be the complete implementation and supplemented with error prevention and -handling features.

On top of this DSL layer, `Processors` foster the handling of temporal dependencies (R_{tmp}) by providing local histories together with an interpolation mechanism as a counterpart to state variable histories stored by the state variable's managing actor (Wiebusch, 2016, pp. 168–170). Listing 16 presentes the implementation of these features. The semantic entity references `RightHand` and `LeftHand` are assumed to be defined in analogy to the previous examples. Furthermore, the `Processor` trait is assumed to store histories of entity property values by means of an extension of the `SemanticValue` base class (line 10). The example actor (lines 1–5) uses the extended access mechanism to acquire the `Position` property of the left- and right hand entity to invoke the semantic function `-`. However, instead of using the local storage access mechanism as shown before, `at Milliseconds(0)` is appended (lines 3–4). This results in the two properties being retrieved with the exact same timestamp, which is different from accessing the latest value of each if they are updated at different intervals. The specified

```
1  class FeatureExtractor extends Processor {/*...*/
2    Updates the properties of entity describedBy RightHand 'with' {/*...*/
3      (Position of RightHand at Milliseconds(0)) -
4        (Position of LeftHand at Milliseconds(0))
5  }}
6  trait Processor extends SimulatorXActor with SemanticEntityReferences {/*...*/
7    implicit var sync: SyncTime = {/*...*/}
8  }
9  abstract class SemanticValue[T] {/*...*/
10   def appendToHistory(value: T, timestamp: Long) = {/*...*/}
11   def at(t: Milliseconds)
12         (implicit sync: SyncTime, interpolator : Interpolator[this.type]) =
13     {/* Applies the interpolator and returns a semantic value with no history */}
14 }
```

Listing 16. Handling of temporal dependencies using local histories and an interpolation mechanism. Refer to the text for details.

time value is interpreted with respect to the latest timestamp at which data for all required entity properties is available within the local storage. *Milliseconds(0)* denotes exactly this latest timestamp; values greater than zero denote earlier points in time. If a property value with a required timestamp is not available in the local data, a value is interpolated using the two nearest available values. The implementation of this access is showcased in lines 6–14. The `Processor` trait defines and updates an `implicit` value representing a special point in time (`sync`, line 7): the latest point in time at which a corresponding value for each required property is present in the local storage. This is the latest point in time at which required entity properties can be accessed using `at` and with respect to which the time value passed to `at` is interpreted. Updating `sync` accordingly guarantees that any access either directly hits a stored timestamp-value pair or hits a point in time for which an earlier and a later stored timestamp-value pair exist. Scenarios where extrapolation would be required are excluded. The counterpart to the `implicit` value `sync` is the extension of the `SemanticValue` base class (lines 9–14) with the method `at` (lines 11–13). Besides a time value, it specifies two implicit parameters: the said synchronization time and a suitable interpolator (line 12). Similar to semantic functions, interpolator implementations are meant to be defined as `implicit` in the actor's scope. The `Processor` does so for both, semantic functions and interpolators, using default implementations. Inheriting classes still may define refinements.

The `Processor` implementation is complemented with the `Producer` trait that can be used as clock generator for repeating tasks, like animations, as well as for testing processors. Listing 17 showcases the `Producer` trait as well as an auxiliary DSL syntax for correctly starting actors (akka actors may not be directly created using `new`) and for creating simple entities. The semantic entity reference `RightHand` is assumed to be defined in analogy to the

```
1  Start a new Producer { //that
2    Is named "Rotator"
3    Creates entity `with` properties RightHand named "RightHand"
4    Updates the properties of RightHand every Milliseconds(16) `with` {
5      val rawAngle: Float = math.radians(Context.elapsedTime / 10f)
6      Angle(rawAngle)
7  }}
8  Start a new Processor { //that
9    Requires property Angle from RightHand
10   Updates the properties of RightHand `with` {
11     val rawAngle    = (Angle of RightHand).value
12     val rawPosition = Mat.rotateZ(rawAngle) * Vec3.UnitX
13     Position(rawPosition)
14 }}
```

Listing 17. A simple `Producer` used to supply a `Processor`.

previous examples. A `Producer` is defined in lines 1–7 and started using the `Start a` DSL syntax, which wraps akka's actor bootstrapping. It is assigned the name *Rotator* for debugging purposes (line 2). Upon creation it creates an entity and assigns the properties specified by the semantic entity reference `RightHand` to it (line 3), instead of using the properties to query a matching entity. Lines 4–6 define a processing rule similar to the `Processors` shown before. Its updates the `Angle` property of the entity (described by) `RightHand` with a constantly increasing value. However, this rule is not triggered by a value change of a required property, but rather every 16 ms. The truly elapsed time is accessed using the `Context` data structure, defined and updated by the `Producer` base class. The `Processor` defined in lines 8–14 reacts to the value changes caused by the `Producer`. It requires the `Angle` property from the `RightHand` entity (line 9). Whenever the value of this property changes, it updates the same entity's properties with a `Position`, following a trajectory on the unit circle around the origin according to the angle's value.

Altogether, the implementation of the decoupling by semantics technique provides processing chain like elements, to which data sinks and sources do not have to be passed upon creation due to the semantic entity reference mechanism. This facilitates their reuse in other applications, as long as the required application state elements exist. A domain specific language layer wraps the underlying state variable access and complements other concepts, like semantic functions, resulting in Scala definitions that read almost as natural language. Since it is defined within the programming language, IDE features, like autocompletion and static code analysis, can be exploited without additional measures. In total, this facilitates API usability. Automatically stored local histories of required properties together with an interpolation mechanism foster the handling of temporal dependencies (R_{tmp}). Traits and implicit parameters are utilized, similar to previously presented implementations, to equip actors with

functionality and conveniently facilitate access to surrounding contexts. The presented version of the decoupling by semantics implementation allows the utilization of semantic entity references that denote exactly one entity. An extension of the DSL for handling descriptions that match multiple entities is an obvious goal for future implementations.

6.3 Multimodal Input Processing

This section showcases the utilization of the reference implementation for multimodal input processing. With the aid of the interaction use case, exemplary processing steps at data-, feature-, and decision level are presented while emphasizing facilitations. Multimodal processing methods that have been implemented to validate the underlying system core are thus outlined. Jointly, they constitute the multimodal input processing framework *miPro*, integrated into Simulator X.

Data-Level

At data level, higher-level information, relevant for analyzing the user's behavior, has typically to be extracted from raw sensor data. The latter is obtained via drivers or SDKs and integrated into the semantic entity-component state as proposed in Figure 5.3. In the context of the interaction use case, this could be the position and orientation of the user's hands and head, represented as transformation matrices obtained via a tracking system, as well as the user's utterances, represented as string tokens obtained via a microphone and preprocessed by an ASR. Simple `Processors` can be applied to extract the position from these matrices and subsequently calculate the hands' velocities, as a basis for an gesture detection step. Listing 18 showcases such a simple `Processor` (lines 2–6). It requires the `Transformation` property of the entity described by the semantic entity reference `RightHand` (line 3, cf. line 1), which represents the user's right hand. It updates the properties of this entity with the `Position` extracted from the required `Transformation` using a semantic function, whenever the `Transformation` changes (lines 4–5).

```
1 val RightHand = Type(hand) and Chirality(right)
2 Start a new Processor { //that
3   Requires property Transformation from RightHand
4   Updates the properties of entity describedBy RightHand `with` {
5     Position from (Transformation of RightHand)
6 }}
```

Listing 18. A simple `Processor` for feature extraction at data-level.

The listed definition of the `Processor` almost matches the description proposed in Figure 5.12, while still satisfying the native Scala syntax. Some Scala particularities, however, are still visible, e.g., the fact that `describedBy` can not be written apart in this position and that `with` has to be surrounded with quotation marks to not conflict with the eponymous Scala keyword.

Listing 19 showcases a second `Processor` that calculates the hand's velocity from the extracted `Position` as well as an additional menas to foster reuse. Since the `Processor` DSL is Scala internal, common means for generalization can seamlessly be applied. In this case, the frequently required operation of calculating a velocity from a trajectory of positions is bundled in a specialized `Processor`, named `VelocityProcessor` (lines 3–14). It is still compatible with other DSL parts, i.e., it can be started as any other `Processor` (line 1). Its implementation specifies a semantic entity reference to be passed (line 4) as well as an optional time delta (line 5) for the actual calculation. It requires the `Position` of the referenced entity (line 8) and updates its properties with a `Velocity` property, whenever the `Position` changes. As counterpart to the time delta, a position delta is calculated using the local storage history access mechanism to subtract a `Position` value in the past (line 11) from the latest available `Position` value (line 10). The new `Velocity` is then the position delta divided by the time delta (line 12). `Processors` defined that way, encapsulate calculation rules and hence avoid redundancy. A collection of such definitions is a means to further foster the reuse of common operations and provides developers with an appropriate toolset for data-level processing.

```
1  Start a new VelocityProcessor(RightHand)
2
3  class VelocityProcessor(
4    entityWithPosition: SemanticEntityReference,
5    deltaT: Milliseconds = Milliseconds(60)
6  ) extends Processor
7    {
8    Requires property Position from entityWithPosition
9    Updates the properties of entity describedBy entityWithPosition ‘with‘ {
10     val deltaP = (Position of entityWithPosition at Milliseconds(0)).value -
11                  (Position of entityWithPosition at deltaT).value
12     Velocity(deltaP / deltaT)
13   }
14 }
```

Listing 19. A generalization mechanism for `Processors`.

Feature-Level

At feature-level, relevant features are assumed to be extracted. Processing steps typically transfer multiple relevant features to more meaningful, e.g., symbolic, representations. For this purpose, specialized `Processors` can be applied to wrap complex algorithms, like machine learning methods, that are often implemented within external libraries. Listing 20 showcases this application by a simple gesture detection based on neural networks. Lines 1–5 define a specialized `Processor` (`SupervisedLearningProcessor`) that performs a classification task. It is described by largely using the standard `Processor` DSL: it requires the `Position` and `Velocity` property of the entity described by the semantic entity reference `RightHand` (line 3) and it updates the `PointingConfidence` property of the entity described by the semantic entity reference `User`, whenever one of the requirements change (line 4). The additionally required configuration of the underlying neural network is outsourced into an external file. `SupervisedLearningProcessor` extends the DSL with a new statement for its specification (line 2). Similarly, `SupervisedLearningProcessor` defines a method that triggers the evaluation of the neural network (`prediction`), which constitutes the processing rule (line 4). The inputs to the underlying network are automatically fetched from the required properties. `SupervisedLearningProcessor` constraints the `Requires` DSL syntax to only accept semantic types of floating point-based data types. All requirements are then sorted, rolled out if necessary (e.g., matrices are converted to float arrays), joined to one float array, and used as input to the network. A minimum configuration thus gets along with the definitions shown in the listing. In the context of the example, a second specialized `Processor` classifying rotational gestures is assumed to be analogously defined. A simple `Processor` can then be used to finalize the classification by selecting the gesture with the highest activation value and by subsequently updating an entity property

```
1  Start a new SupervisedLearningProcessor { //that
2    Is configuredBy NeuralNetworkConfiguration("pointing.xml")
3    Requires properties (Position and Velocity) from RightHand
4    Updates property PointingConfidence of entity describedBy User `with` {prediction}
5  } // Assuming a similar processing step for 'rotate' gestures
6  Start a new Processor { //that
7    Requires properties (PointingConfidence and RotateConfidence) from User
8    Updates the properties of entity describedBy User `with` {
9      if((PointingConfidence of User) > (RotateConfidence of User)) Gesture(pointing)
10       else Gesture(rotate)
11   }}
```

Listing 20. A specialized `Processor` realizing a simple pattern recognition by means of supervised learning methods (revised from Fischbach, Wiebusch, & Latoschik, 2017).

that represents the currently performed gesture (lines 6–11). This `Processor` requires the confidence properties yielded by the `SupervisedLearningProcessors` (line 7). Whenever one of the confidences change, it updates the `Gesture` property of the entity `User` with a grounded symbol representing the gesture with the highest confidence value (lines 9–10).

The showcased `SupervisedLearningProcessor` implementation wraps the Java machine learning framework *Encog* (Heaton, 2015). It comprises automatic configuration strategies, e.g., for network topologies and input mapping, facilitating concise definitions, like the one shown in this section. It is complemented with dedicated actor implementations and tool support that provide means to record, annotate, and playback entity property changes—a necessity for the training process of supervised learning methods. Altogether, it constitues a further building block in the repertoire of *miPro*.

Decision-Level

At decision-level, *miPro* provides two multimodal fusion methods: a tATN (Zimmerer et al., 2016; Zimmerer, 2016) based on Latoschik's (2002) proposal as well as an unification approach (Link, 2017) based on Johnston's (1998) proposal. Their reference implementations utilize the underlying semantics-based techniques of Simulator X to realize aspects of the fusion methods if possible.

Figure 6.4 illustrates the concept of the tATN applied to the interaction use case. A tATN definition comprises states (circles), transitions (black arrows) with constraints, and functions that are executed if a transition is carried out (white arrows). Cursors (green) represent active interim results. They fill their associated register (orange) as they are traversed through the tATN: relevant application state changes, e.g., speech recognition- and gesture detection updates, trigger the evaluation of the cursors' outbound transitions. Reaching an end state (not shown) corresponds to a successful parse of a multimodal utterance and is assumed to gather all data relevant for the invocation of an appropriate grounded action. Due to the explicit definition of transition functions, the application state can be altered at any point of analysis to give feedback to the user, e.g., by utilizing a `Processor` to continuously map posture features to the chair's orientation (as required by the interaction use case).

Semantics-based techniques are incorporated as follows: Registers are proposed as key-value storages for interim results by the aATN concept. They can simply be implemented as a map using string identifiers. *miPro*'s tATN, however, uses a `SemanticValueSet` (blue) for the register implementation, providing an interface that is uniform with the application state access and that thus implies no obstructive conversion of representations. Moreover, the general advantages of the semantic grounding and code from semantics techniques,

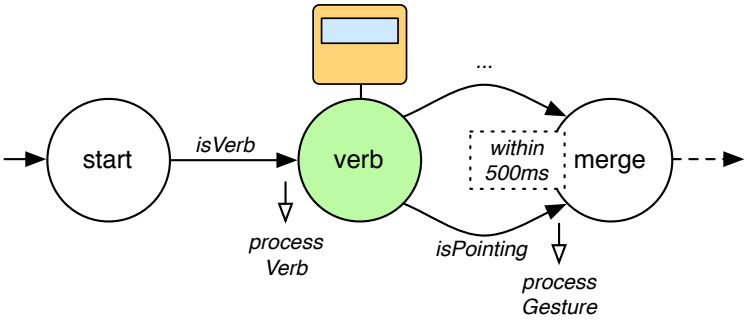

Figure 6.4. Excerpt of a tATN capable of parsing "*Put* [pointing] *that*" (Fischbach, Wiebusch, & Latoschik, 2017, ©IEEE).

like introspection and reasoning support, can thus be exploited by transition constraints and -functions. Semantic queries based on the `Dictionary` shown in Listing 11 as well as grounded actions complement the means for the implementation of transition. Transition functions can hence map triggering state updates to semantic values, entity filters, and grounded actions to store them in their register. A cursor traversal through the tATN thus gathers information related to a user utterance. The structure of the network and the temporal constraints specify syntactic- and temporal correctness (R_{syn} and R_{tmp}), respectively. Semantic correctness (R_{sem}) has to be validated based on the contents of the register. Grounded actions are a fitting means for realizing such a validation, especially in instruction-based scenarios as proposed by the interaction use case. Utterances in imperative mood start with a verb that can be mapped to a grounded action. This action is stored in the cursor's register. Its parameters serve as a frame that has to be completed during traversal, e.g., the `collocate` action requires two entities. Its preconditions serve as semantic constraints that have to be satisfied. Finally, a successful traversal results in the execution of the action.

An alternative method for the joint analysis of multimodal input at decision-level provided by *miPro* is unification-based fusion. In the context of this method, all relevant information is represented by so called *feature structures*. A central unification operation combines compatible feature structures and thus gathers all data relevant for the invocation of an appropriate grounded action, while performing syntactic-, temporal, and semantic constraint checks (R_{syn}, R_{tmp}, and R_{sem}). Special feature structures serve as targets for the unification process, representing a valid user command with most properties being undefined and meant to be completed by unification operations (see Figure 6.5). A completed target feature structure corresponds to a successfully parsed utterance and implies the executing of a corresponding action. Feature structures basically are key-value sets, where values are either atomic or another feature structure. The utilization of semantic types and -values for an implementa-

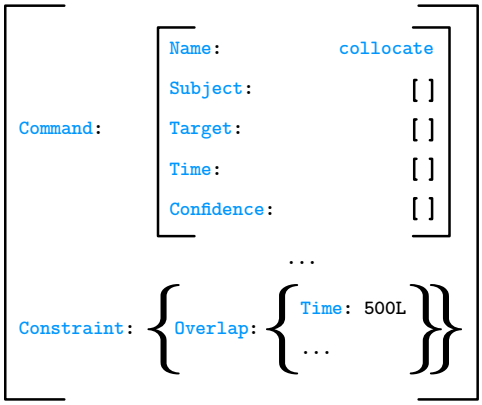

Figure 6.5. A feature structure excerpt serving as target for parsing "*Put* [pointing] *that* [pointing] *there*". The underlying key-value concept is implemented by using semantic types (blue) as keys and semantic values as values. `Command` represents one valid user utterance and corresponds to an action referenced by the grounded symbol `collocate`. All other properties of `Command` are still undefined. Additional contraints, relevant for the fusion process, e.g., the temporal dependency between pointing gestures and determines (curly braces), extend the feature structure concept (as proposed by Johnston, 1998).

tion results in benefits similar to those of the tATN: uniformity to the application state access interface and introspection as well as reasoning support.

Besides this fusion method specific utility, both implementations benefit from semantics-based techniques as follows: `Processors` are an adequate means for the execution of the fusion method. Relevant application state changes can then be specified by defining `Processor` requirements, using semantic entity references as well as the DSL. The implementations of the fusion methods thus are decoupled by semantics, just as the specialized `Processor` implementations shown before. In addition, both implementations may use semantic queries and apply reasoning to resolve referents in the user's utterance (cf. section 5.4). For instance, a transition function of a tATN could use all semantic values gathered in its associated register to create an entity filter that matches all entities that are of color green and are near a given pointing ray.

6.4 Ancillary Contributions

This section briefly describes ancillary software parts that have been developed in the context of this thesis to facilitate the implementation of proof of concept demonstrations as well as to ease the utilization of Simulator X in student projects and lecture modules.

Subsystems

The subsystems developed for Simulator X include a runtime editor for entity properties, input integration subsystems, a physics simulation subsystem, as well as a subsystem that facilitates the creation of 2D graphical user interfaces. In addition to that, Simulator X comprises further subsystems, predominantly developed by other authors, like for 3D rendering, for sound rendering, for the integration of various input devices, for the integration of the Virtual Reality Peripheral Network protocol (VRPN Taylor et al., 2001), and for incorporating the game engines Unity 3D and Unreal Engine 4.

The editor subsystem provides a graphical user interface allowing to inspect and alter the properties of all entities in the application state during runtime. During startup, the subsystem asks the central registry to be notified upon any entity creation. Whenever it is notified, it starts observing all of the entity's properties and stores them locally. User's can browse the editor's interface and select entity properties. The views that are used to present and set properties are selected based on the property's semantic type. For instance, if a user selects a `Color` property, she can choose amongst all views able to present values of data type `Vec3`. This selection may include a view simply displaying three floating point numbers as well as a view interpreting the `Vec3` as a color and using it to fill a square. Once chosen, a view is utilized to visualize all entity properties of the same semantic type in future inspections. Entity properties are hence distinguished from other entity properties that have the same data type by means of the semantic grounding technique. The same mechanism is applied for user interface elements dedicated to setting an entity property. Both, setters and views can be extended by implementing an automatically-built skeleton, which can be altered even at runtime. Altogether, the editor subsystem provides a means for inspecting and altering the application state that is handy for debugging and is independent of concrete entity property types, due to mechanisms applied for selecting and extending views and setters.

The developed input integration support comprises two subsystems. One wrapping the *leapmotion* SDK (Leap Motion, Inc., 2017) and one wrapping a *TUIO* protocol client (Kaltenbrunner, 2017). The former allows to access the data captured by an eponymous hand tracking sensor. The latter is a protocol to communicate inputs to interactive surfaces, such as touch positions as well as positions and orientations of tangible user interface elements. Both subsystems create and register entities representing the integrated data on startup. After startup, they react to callbacks of the respective libraries, which are called as new input is available. In these callbacks, the obtained information is converted if necessary and integrated into the application state by updating the respective entity properties.

The physics simulation subsystem wraps the Java library *JBullet* (Dvorak, Martin, 2017).

The implementation provides components that can be used for marking entities to be simulated as rigid bodies, including spheres, boxes, and cylinders. Such components are typically combined with components representing textured 3D meshes, provided by rendering subsystems, to create entities that account for the visible part of virtual environments. Content creation is facilitated by enabling the definition of rigid body shapes by means of the 3D asset exchange schema *COLLADA* (Khronos Group, 2017).

The creation of 2D graphical user interfaces is facilitated by a subsystem that serves as an abstraction layer on top of basic 3D rendering functionality. For this purpose, the implementation provides components representing common 2D GUI elements, like buttons, labels, and icons. These components require to be combined with a 3D mesh component for entity creation. Thus created entities possess two subsets of properties: one describing the 2D GUI element, comprising 2D positions and orientations as well as texts or images, and one describing an underlying textured 3D quad. The former is the interface to application developer. The latter is used by the 2D GUI subsystem to map changes within the 2D element properties to the 3D quad. For instance, if the text and 2D position of a label is updated, the 2D GUI subsystem generates a texture depicting this text, calculates the correct transformation matrix for the quad to appear correctly on the screen (with respect to the camera position), and sets the corresponding entity properties, e.g., `Transformation` and `Texture`.

Application Layer

Bootstrapping, necessary for launching Simulator X applications, is gathered as a set of traits and convenience functions. These tools jointly provide a layer that facilitates the development of applications by hiding details predominantly relevant for subsystem or core development. Listing 21 showcases the application layer by means of a minimal example. It consists of a general entry point (lines 1–2) as well as an actor that is dedicated to spawn required subsystem actors and run application logic (lines 4–11). The trait `SimulatorXApplicationMain` (line 2) encapsulates the startup procedure for the first `SimulatorXActor` and provides the actual `main` function. It finally spawns one instance of `ProducerProcessingExample`. This actor facilitates the definition of a set of subsystems that realize the ECS interface and potentially provide components (lines 5–6, cf. Wiebusch, 2016). Upon its startup, the thus defined subsystems are properly created and registered, including the awaiting of asynchronous bootstrapping on their part. After this process is finished, the callback `finishConfiguration` is invoked, allowing the application developer to create entities as well as `Processors` (lines 7–10). The parameter `subsystems` (line 7) allows to send messages to counterpart actors of

```
1  object ProducerProcessingExample extends
2    SimulatorXApplicationMain[ProducerProcessingExample]
3
4  class ProducerProcessingExample extends SimulatorXApplication {
5    def applicationConfiguration =
6      ApplicationConfiguration withSubsystem Editor(name = "editor")
7    def finishConfiguration(subsystems: Map[String, SimulatorXActor.Reference]) {
8      val RightHand = Type(Symbols.hand) and Chirality(chirality.Right)
9      /* Create entities and processors */
10   }
11 }
```

Listing 21. A minimal example for creating an application within the Simulator X platform.

created subsystems, since it contains a mapping between specified name (e.g., `"editor"`, line 6) and actor references. `finishConfiguration` could in this case consist of the `Producer-Processor` pair shown in Listing 17. The editor subsystem would then allow to inspect the continuously changing values. Developers that are unexperienced with Simulator X would thus be supported in comprehending the example, at best by also altering it and observing the consequences in the editor's GUI.

Code Examples

Supporting novice developers with examples is in general an effective means for assisting them in learning a new API (McLellan et al., 1998). The presented (specialized) processors as well as the application layer are thus accompanied by code samples, which are briefly listed in the following. A package of basic examples introduces to the application layer and showcases some common subsystem configurations, including the combination of 3D rendering and physics simulations as well as the combination of 3D rendering and 2D GUI subsystem. A second set of code samples deals with `Processors`, `Producers`, and the associated DSL. It includes, for instance, the code shown in shown in Listing 17. A third packages is comprised of multimodal input processing examples utilizing simple and specialized processors. It includes configurations suitable for recording training data, annotating, and training a neural network (cf. Listing 20) as well as simple decision-level fusion scenarios using the tATN implementation.

6.5 Summary

This chapter presented the reference implementation of the semantics-based software techniques proposed in chapter 5. The taken design decisions as well as the key mechanisms

applied for the implementation resulted in several advantages in addition to those inherent to the techniques (see Table 6.1, design decisions). The actor model provides a scalable communication-, execution-, and distribution scheme, fostering modularity and reusability. The choice of OWL guarantees compatibility to a widespread set of existing ontologies and tools. The primary programming language Scala provides adequate means for realizing usable APIs, due to its functional aspects and its syntax flexibility. Scala is platfom independent and compatible with a large number of existing software libraries, since its programs are compiled to bytecode for the *Java Virtual Machine*.

The emphasized key mechanisms serve two kinds of purposes (see Table 6.1, mechanisms). Some realize or extend functionality that is conceptually demanded by the semantics-based software techniques. State variables provide the illusion of global variables in accordance with the actor paradigms (R_{sta}). Semantic entity references, in combination with singleton actors and context binding, render the explicit reference passing unnecessary and decouple subsystems by means of semantic descriptions (R_{acc}). The local storage of histories of required entity properties in combination with an interpolation mechanism provides additional means for temporal synchronization (R_{tmp}).

Other key mechanisms foster API usability. Traits, implicit parameters, and Scala's `object` construct are utilized to maintain concise syntax when using the core API. The internal DSL makes definitions read almost as natural language descriptions while exploiting IDE support, like autocompletion and static code analysis. Tool support and development process integration furter fosters API usability by reducing the development effort.

The implemented repertoire of common multimodal input processing steps, adding up to the *miPro* framework, facilitates the creation of multimodal interfaces. The implementation

Table 6.1. Taken design decisions (DD) and key implementation mechanisms mapped to their contributions to maintainability as well as to solve the fundamental (I)RIS requirements identified in chapter 2.

	Applied measure	Contributes to
DD	Actor Model	R_{exe}, R_{com}, modularity, reusability
	OWL	R_{per}, API usability
	Scala	API usability
Mechanisms	State variables	R_{sta}
	Semantic entity references, context binding, & singleton actors	R_{acc}
	Local histories & interpolation	R_{tmp}
	Internal DSL, tool support, implicit parameters, Scala's `object` syntax, delimited continuations, currying, & traits	API usability

of those processing steps showcases the application of Simulator X's core API. Advantages arise especially due to the uniform access scheme and the use of semantic queries for referent resolution during decision-level fusion. Two reference implementations of such fusion methods are provided, one based on tATNs and one based on unification. Both can be applied to check intentional multimodal inputs with respect to its syntactic-, temporal-, and semantic correctness (R_{syn}, R_{tmp}, and R_{sem}). They benefit from an enhanced compatibility to the system's core API, since their implementation is based on the semantic grounding technique as well. Simulator X and *miPro*, however, are not limited to this use case. `Processors`, for instance, can also be utilized to realize multimodal output generation steps as can feature extraction and classification steps be applied to unintentional non-verbal input.

The presented ancillary contributions complement the Simulator X platform by adding functionality supporting development. Moreover, the abstraction layer for application development as well as the code examples facilitate the utilization of Simulator X for teaching modules and student projects.

Chapter 7

Validation and Method Exploration

This chapter summarizes the results of all methods applied to validate functional and non-functional software quality aspects of the presented semantics-based techniques as well as of their reference implementation. Moreover, it presents all activities conducted to explore the assessment and improvement of IRIS maintainability.

Expert reviews are chosen as primary method since they are most reasonable and the most commonly applied form of evaluation (see section 3.1). Proof of concept demonstrations, pre-studies, and informal evaluations, the three (other) validation methods commonly applied for MMSs (see section 3.2) and RISs (see section 3.3), are utilized as follows. Several proof of concept prototypes practically validate feasibility. Two pre-studies explore appropriate methodologies for assessing long-term API usability to counter the lack of effective methods identified in section 3.1. They yield helpful best practices for future full studies. Finally, informal insights gathered throughout the design and development are summarized as *lessons learned* to support future development in the area.

In addition to the presented validation results, other authors specifically analyze the techniques as well as the reference implementation with respect to reusability (Wiebusch, 2016, pp. 185–208) and with respect to the application of the actor model with respect to performance, latency, and concurrency (Rehfeld, Tramberend, & Latoschik, 2013, 2014, 2016).

7.1 Expert Reviews

Expert reviews are subjective assessment methods that are based on the judgment of professionals and recorded on paper (Riaz et al., 2009; McIntosh et al., 2016). Contributions to workshops, conferences, and journals are written by experts and reviewed by other experts in a scientifically established process. Successful publications that analyze non-functional IRIS qualities, like maintainability and API usability, thus constitue a valid and counterchecked form of an expert review; including this very thesis. All conducted reviews of that kind, as-

sessing the reference implementation and the semantics-based techniques, are summarized in the remainder of this section.

Scala, Actors, & Ontologies

The three design decisions taken for the reference implementation (the programming language Scala, the actor model, and OWL ontologies) as well as the code from semantics technique are evaluated by Wiebusch, Fischbach, Latoschik, and Tramberend (2012) with respect to reusability and modifiability.

Drawn conclusions

The application of the actor model is found to be beneficial in terms of scalability and maintainability. Scalability is credited due to its support for intra- and inter-node concurrency and flexibility in terms of actor granularity. Resulting in an effective utilization of available hardware resources. Maintainability is credited due to its uniform message interface, thanks to which the number of different APIs that have to be maintained is reduced.

The code from semantics technique is concluded to foster decoupling and to increase reusability and modifiability. KRLs are a basic requirement for realizing IVEs that is commonly implemented by loosely coupling an ontology. Although such an approach is already beneficial in terms of reusability, it is restricted to the functionality accessible through the API of the underlying (I)RIS. The core-level integration proposed by the code from semantics technique, however, minimizes the access restrictions between KRL and platform core. It fosters uniformity as well as introspection (semantic reflection) on all layers and hence enhances decoupling and modifiability. Moreover, loosely coupled solutions often lead to redundant definitions, since each subsystem may use its own ontology. The uniformity of the code from semantics technique implies coherence and avoids the necessity of ontology synchronization.

Finally, abstraction mechanisms provided by a programming language as well as the suitability of its syntax to write concise definitions are identified to be of equal importance to an IRIS implementation as performance is. The review concludes that Scala facilitates reusability and modifiability and emphasizes the utility of its support for internal DSLs as well as its functional aspects.

Adequacy for Multimodal Processing

A subset of the techniques, design decisions, and implementation mechanisms presented in this theses are analyzed by Latoschik and Fischbach (2014) with respect to multimodal

processing: semantic grounding, code from semantics, and decoupling by semantics, as well as the actor model, semantic functions, and the internal DSL.

Drawn conclusions

The reference implementation's underlying communication- and execution scheme , the actor model, is identified to be an adequate means for coping with the concurrency inherent to multimodal utterances. The actor model's s flexibility in granularity and the concomitant scalability are the most important aspects supporting this adequacy.

The utilized DSL is concluded to reduce the diversity of programming styles and thus to increase code quality. The realization of the DSL within the Scala language, allows to exploit type- and syntax checks and thus further fosters API usability.

Semantic functions contribute to code quality, as they keep the DSL clean. In addition, they pose a handy means to deal with variance due to individual user characteristics and cultural context. With regard to multimodal processing, this implies that operations of the same type may have to be handled differently with respect to the semantics of their operands. For instance, equality between to time stamps relating speech and gestures may still be true within a time window of several milliseconds, whereas other comparisons require true equality. This time window may vary between individuals and cultural context. Semantic functions allow to adequately formalize this flexibility, especially due to their support for binding alternative implementations.

The semantic grounding- and code from semantics techniques add to the internal DSL mechanism by providing two of the DSL's building blocks: semantic types and -values. In combination with semantic functions, they facilitate semantic correctness on API level. Semantic functions take semantic values as parameters. A `Radius` parameter, for instance, can thus not be filled with a `Diameter` value (in contrast a plain floating point parameter). On an algorithmic level, semantic grounding and code from semantics facilitate semantic correctness as well, by enabling the use of reasoning methods, e.g., during multimodal fusion.

All these measures add up to an early version of the decoupling by semantics technique, which is identified to be a customizable and reusable technical solution for reoccurring multimodal processing tasks.

Semantic Entity-Component State Access and Grounded Actions

Fischbach, Wiebusch, and Latoschik (2016) analyse the techniques *semantic entity-component state* and *grounded actions* with particular regard to the coupling dilemma and thus to maintainability.

Drawn conclusions

The definition of entities and actions by means of semantic descriptions is identified to eases their reuse in different multimodal applications. Semantic grounding fosters flexibility, since it allows to specify based on semantic concepts. Underlying data types can be altered requiring no changes in high-level DSL descriptions. This process can be additional eased by utilizing automatic type converters (as proposed by Wiebusch & Latoschik, 2012).

Grounded actions provide flexibility, especially in combination with the code from semantics technique. For instance, verbs associated with an action can be easily adapted in the ontology to extend the utterances an interface accepts, without requiring in-code changes. In total, those benefits are concluded to support modifiability and reusability.

The code from semantics technique itself, is analyzed to further facilitate reuse. One the one hand, by enabling the use of existing tools and, on the other hand, by being an programming language independant abstraction layer. The former is especially beneficial for multimodal processing since it includes reasoning software. The latter enables the reuse of concepts between (I)RIS platforms.

Altogether, the analyzed techniques are concluded to realize close semantic subsystem interrelations without implying close coupling. They are hence a suitable foundation for IRISs realizing multimodal interfaces, which facilitates maintenance.

All Techniques and Complete Reference Implementation

Fischbach et al. (2017) present a comprehensive analysis of the techniques presented in this thesis and reflect on the complete reference implementation with respect to modularity, modifiability, reusability, and API usability. The semantic entity-component state technique is presented as part of the semantic grounding technique in their contribution.

Drawn conclusions

The semantic grounding technique is analysed to separate the agreement on identifiers and their meaning from their use in APIs and to decouple it from utilized data models. Moreover, it provides introspection capabilities independant of the features of the target programming language. The ECS pattern's component mechanism, which lacks this feature, is thus improved by.

Grounded actions are concluded to provide additional flexibility for the definition of behavior, due to their compatibility with common planning approaches, like PDDL, and the consequential possibility of specifying system behavior in a declarative way. The uniform interface amongst application state, grounded actions, and ontology-based planning (due to the

semantic grounding- and code from semantics technique) is especially beneficial for maintenance. In all, grounded actions decouple the description of operations that are triggerable via the user interface, their implementation, and their execution. They thus complement the application state concept of the ECS pattern by providing a means to define reusable behavior. Moreover, they are are especially beneficial for multimodal processing, since they constitue programmatic counterparts to command a user shall be able to trigger.

Semantic queries are analysed to facilitate the lookup of entities based on semantic descriptions as well as on numeric properties. They realize a feature set commonly required by AI methods, while avoiding the parsing overhead inherent to externalized approaches based on semantic query languages, like SPARQL. The technique is used to extend the ECS pattern with a mechanism, allowing subsystems to access entities that are not associated with one of its supported components.

The code from semantics technique is reviewed to improve interface definitions by increasing cohesion as well as decoupling and by facilitating their reuse even for other (I)RISs. The associated code generation approach entails fast state access at runtime, while maintaining the possibility to access the ontology. Potential development overhead is countered by dedicated editing software. Its development is eased if existing ontology tools can be utilized. The potential to use reasoning software allows the enrichment of the application state with inferable facts and facilitates the implementation of semantic constraint checks during multimodal fusion.

Decoupling by semantics is analysed to decouple subsystems in terms of data sinks and sources, which are then highly reusable. The utilized high-level API is reviewed to foster usability for developers. In this context, Scala proves to be an beneficial choice that enables the language-internal definition of this API and thus makes common IDE features exploitable.

Altogether, the semantics-based techniques are concluded to improve the ECS pattern by extending introspection capabilities and by proposing a mechanism for inter-system reuse. In addition to revising the uniform semantic access scheme to the application state, they extend the pattern by a technique for semantically describing behavior. They are thus a suitable basis for implementing complex AI dependent RISs, e.g., realizing multimodal interfaces.

The design decisions that underly the reference implementation are reviewed as follows. The actor model fosters scalability and extensibility while Scala's syntactical flexibility facilitates the definition of APIs. The usability of those APIs is evaluated and improved by the application of the API peer review method proposed by Farooq and Zirkler (2010). The results of this process constitute the code samples presented in the contribution (and in this thesis). Readers can follow this review's rationale and assess the API usability themselves on the basis of those samples and the corresponding explanations.

Summary

Jointly, the four presented expert reviews assess all six semantics-based techniques, all three design decisions, and implicitly most implementation techniques, since they include an analysis of the reference implementation's API. The drawn conclusions state that the techniques solve the ECS pattern's runtime type deficit, improve component granularity, facilitate access to entity properties outside a subsystem's component association, incorporate a concept to semantically describe behavior as complement to the state representation, and enable compatibility even between IRISs. Furthermore, the three design decisions are assessed to meet the identified functional requirements while fostering software quality. The actor model provides a scalable communication-, execution-, and distribution scheme. The choice of OWL guarantees compatibility to existing ontologies and tools. Scala facilitates realizing concise APIs, is platfom independent, and compatible with a large number of existing software libraries. Ultimately, the four expert reviews confirm the improvement of maintainability.

7.2 Proof of Concept

This section presents the most relevant implemented proof of concept demonstrations. These demonstrations validate feasibility and the completness of captured requirements to support the theoretical assessment of software qualities by means of expert reviews. Their implementation accompanied the development process of the presented semantics-based techniques as well as of Simulator X and supported the assessment and improvement of the process itself. In the following, the demonstrations are briefly summarized by highlighting aspects that are primarily validated (see Fischbach, Wiebusch, Rehfeld, Tramberend, & Latoschik, 2016, for illustrating media).

SiXton's Curse

SiXton's Curse (Fischbach et al., 2011) is the first demonstration of the Simulator X platform. It validates the feasibility of the semantic grounding technique, the design decisions Scala and the actor model, as well as the implementation mechanism *state variables*.

The demonstration realizes a virtual medieval village that is inhabited by ghosts. The user takes the role of a wizard, can explore the village, and cast spells to prevent the ghosts from destroying it via a multimodal interface. Technically, it utilizes subsystems for input integration, physical simulation, 3D graphics- and sound rendering, path planning, as well as multimodal processing. The demonstration is meant to run in a CAVE-like (Cruz-Neira, Sandin, DeFanti, Kenyon, & Hart, 1992) environment, equipped with sensors that enable to

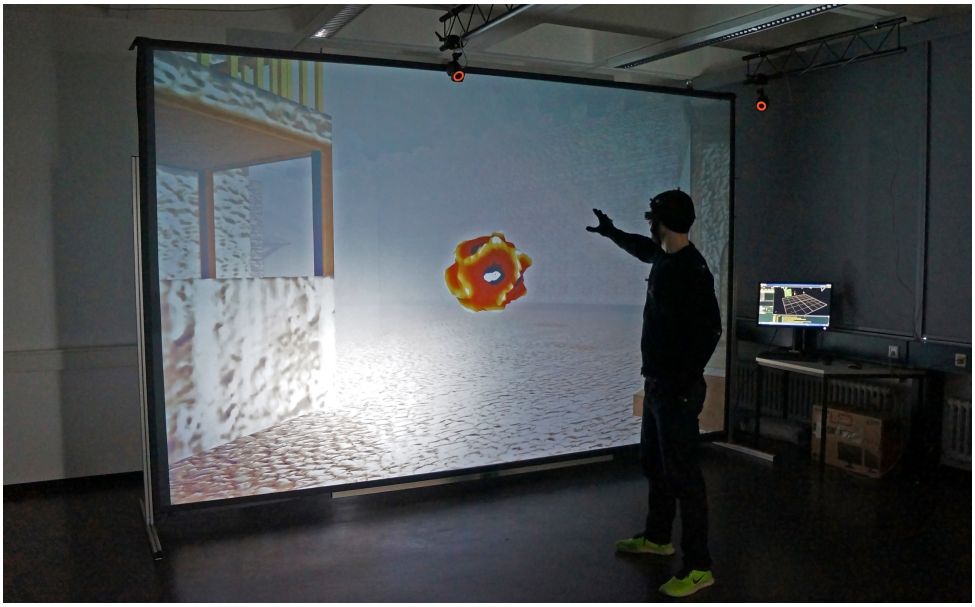

Figure 7.1. The *SiXton's Curse* demonstration. A user casting a *fireball* spell by uttering "*Summon* [hands together in front of the body] *purgatory* [push gesture]."

track the position and orientation of (at least) the user's head and hands as well as to record her voice (see Figure 7.1).

smARTbox

The *smARTbox* setup is the technical basis for a series of four MR demonstrations developed using Simulator X. The first two demonstrations in the series (Fischbach et al., 2012b; Fischbach, Latoschik, et al., 2012) test the performance and scalability of the actor model in combination with the state variable mechanism by means of a school behavior simulation of fishes. The third demonstration (Fischbach et al., 2012a) validates high throughput for entity property changes. The forth demonstration (Fischbach, Treffs, et al., 2012) showed that Simulator X can be effectively used by novice (I)RIS developers by exploring application areas of the *smARTbox* setup within a student project.

The setup itself is a low-cost interactive surface providing capabilities for stereoscopic projection, audio feedback, touch- and fiducial marker detection, as well as markerless tracking of the users' upper body (Figure 7.2). The school behavior is implemented (based on Reynolds, 1987) so that the mapping between fishes to be simulated and actors performing simulations can be configured. At the extreme, this can be one-to-one mapping caus-

Figure 7.2. The *smARTbox* demonstration. A user interacting with a school of virtual fishes via a tangible interface based on the *Window on a World* metaphor (Feiner, MacIntyre, Haupt, & Solomon, 1993). The perspective was registered with the camera for the photo. ©ACM

ing a high level of communication and forcing the actor scheduler to perform many context switches. The third demonstration in the series extends the virtual fishtank by a self-reflection feature, showing a virtual mirror image of the user on the surface. Its implementation uses the video stream from the setup's depth camera to generate a mesh of the user's upper body. This mesh is communicated to the rendering subsystem via a corresponding entity property.

Input Device Evaluations

The adequacy of Simulator X for input processing is validated in the context of two student projects evaluating input devices for VEs (Wiebusch, Fischbach, Strehler, et al., 2012; Fischbach et al., 2013). Fischbach et al. (2013) include multimodal processing in particular by combining the examined tracking devices with speech input in a synergistic manner in one part of the evaluated conditions.

XRoads

The *Cross Reality On A Digital Surface* project (XRoads Fischbach, Zimmerer, Link, Giebler-Schubert, & Latoschik, 2016) aggregates two series of interactive tabletop game prototypes that combine real and virtual elements. The first series validates the feasibility of utilizing

Figure 7.3. The turn-based porting of the *Quest* board game. Up to four players collaborate in a skirmish against another player.

Simulator X for interactive surface environments. This includes the interplay of basic subsystems, like for 3D rendering, the integration of further input device wrappers and auxiliary subsystems, like the 2D GUI subsystem, as well as Simulator X's distribution capabilities. The second series additionally validates the *miPro* framework, including the tATN implementation, for multimodal processing and -fusion as well as the grounded action technique, including the planning subsystem, for executing high-level commands.

Series one is an iteratively extended turn-based porting of the traditional board game *Quest: Zeit der Helden* (Pegasus Spiele, 2014) that explores interaction techniques for this novel kind of games. The initial version (Giebler-Schubert et al., 2013) combines touch and tangibles, i.e., card and pawns, as input modalities. The first extension adds speech and gestures Fischbach et al., 2014. A dice shaking- and throwing-gesture allows the player to roll a virtual dice that bounces off the real pawns standing on the table. A simple speech interface provides an alternative to the existing touch interface for all actions a player can choose from at any point during the game. The second extension adds a mobile device (Zimmerer et al., 2014) that is incorporated into the gameplay by means of a location-based companion

Figure 7.4. The real-time porting of the *Quest* board game. The game master (left) can control one minion manually and all others using high-level commands that are transformed into atomic actions. In this case a succession of move actions. ©IEEE

application (see Figure 7.3). All version utilize Simulator X's distribution capabilities, since speech- and gesture processing run on a separate node for these demonstrations, due to the limited hardware ressources of the utilized interactive surface.

Series two constitutes an alternative approach for porting *Quest* (Link et al., 2016). Instead of creating a faithful copy of the turn-based gameplay, this demonstration exploits the capabilities of the interactive system to realize an collaborative real-time strategy game. The requirement of the game master to control multiple pawns at the same time is realized by implementing less important pawns as completely virtual, autonomous agents, in order to avoid cognitive- and motoric overload. These virtual pawns can be instructed using high-level commands via a speech and gesture interfcae. Such commands are transformed into atomic actions according to the game rules by means of a planning algorithm. Thus the player can still control one tangible pawn manually, while conveniently commanding the rest multimodally (see Figure 7.4).

Big Bang

The *Big Bang* demonstration (Zimmerer, 2016) validates Simulator X's capabilities for realizing multimodal interfaces, with a special focus on decision-level fusion by means of the tATN implementation.

It allows the user to create and manipulate objects in the style of Bolt's (1980) *Put that there*

Figure 7.5. The *Big Bang* demonstration. A user equipped with *HTC*'s *VIVE* hardware setup (foreground) creating, modifying and removing plantes of a solar system (background) via a speech and gesture interface.

(see Figure 7.5). The demonstration can run in two configurations: using a CAVE-like hardware setup including a combination of markerless and marker-based tracking as well as using *HTC*'s hardware setup *VIVE* (HTC Corporation, 2017), i.e., a head-mounted display and two motion controllers. In both configurations the user's utterances are recorded with a clip-on microphone. In the latter configuration, *HTC*'s *VIVE* is driven by the *Unreal Engine 4*, which in turn is loosely coupled to Simulator X (Wiebusch, Zimmerer, & Latoschik, under review, 2017). This demonstration thus showcases that the realization of multimodal interfaces does not conflict with modifiability and reusability qualities of the underlying techniques: the two configurations basically differ just in their subsystem configuration, with no adaptions required to the implementations and definitions that account for the multimodal processing.

Utilization for Teaching

In addition to the validation achieved by proof of concept demonstrations, Simulator X was and is utilized for practical exercises in master level courses (see Fischbach, Wiebusch, Rehfeld, et al., 2016, for selected course results). This implies a certain maturity of the system, since it has to be used by novice (I)RIS developers within a variety of different hard- and software setups. Concrete practical exercise topics included RISs in general, motion analysis for interfaces using machine learning approaches, as well as decision-level fusion for MMIs.

Summary

Altogether, the presented prototypes validate several aspects of the semantics-based techniques and the reference implementation: the interplay of basic subsystems, the distribution capabilities, the techniques *semantic grounding* and *grounded actions*, the design decisions Scala and the actor model, as well as the implementation mechanism *state variables*. Moreover, the *XRoads* series and the *Big Bang* demonstration focus on the *miPro* framework, particularly including the decision-level fusion by means of the tATN implementation and thus implicitly the techniques *semantic queries* and *decoupling by semantics*. Moreover, the reference implementation's API usability is informally assessed in the context of student projects and practical exercises within a variety of different hard- and software setups.

Specific claims about maintainability can not be drawn from proof of concept prototypes alone. Yet, the prototypes indicated issues and positive qualities. For instance, a certain maturity of the reference implementation, since complex applications could be realized even by novice IRIS developers. More importantly, however, the proof of concepts complement the theoretical assessment of maintainability by showing that its practically feasible to meet the necessary functional requirements by applying the presented techniques to an IRIS platform.

7.3 Explorative Studies

Two explorative studies have been conducted in order to identify methods that are reasonable for assessing the API usability of IRIS platforms and to potentially generate hypothesis about platform aspects that influence this software quality. The explorative approach particularly implies that no specific hypothesis are evaluated, yet, it allows to get a better intuition of the usability of the reference implementation's API.

Method

Both explorative studies were carried out in the context of practical exercises to master level courses: a course on machine learning for user interfaces in summer term 2015 ($ML15$) and a course on multimodal interfaces in summer term 2016 ($MMI16$). A comparatively large number of potential participants, i.e., developers, could thus be addressed that were willing to spend a sufficient amount of time to acquire enough experience with the system to solve complex tasks and to reflect on non-functional qualities.

In the context of both courses, participants were given tasks that involved the development of software on basis of the Simulator X platform. These task were meant to be handled in groups of 2 to 3 persons over the course of the whole term, i.e., during six months. In $ML15$

each group had to implement a specialized processor wrapping a neural network, realize a simple gesture detection with it, and utilize it to let a user control a given application. In *MMI*16 each group had to implement a simple multimodal interface in the manner of Bolt's (1980) *Put that there*, using the tATN implementation presented in section 6.3. Students could attend a two-hour practical exercise session per week to get assistance by a supervisor during the lecture time. During the rest of the term, the student groups had time to work on the task on their own. At the end of the term, students that completed the task were asked to answer questions about the API they used via an online survey platform. The survey has been adapted from Wiebusch (2016, pp. 197–206) to facilitate comparisons. It includes the *questionnaire for the subjective consequences of intuitive use* (QUESI, Naumann & Hurtienne, 2010), the *task load index* (NASA-TLX, Hart, 2006), exhaustive additional questions concerning the participants' expertise in programming and RIS, as well as the opportunity to make free form comments.

Results

In *ML*15 14 participants were asked to answer the questions of which 12 replied (2 female, 10 male, aged 21–31). In *MMI*16 17 participants were asked to answer the questions of which 3 replied (1 female, 2 male, aged 24–30). All participants have been notified that the explorative study is not used to assess their personal performance.

All qualitative free form responses have been analyzed and clustered into categories. The two most mentioned negative aspects are the extends of the general documentation (mentioned 7 times) and the amount of code examples (mentioned 3 times). The two most mentioned positive aspects are the decoupling by semantics concept (mentioned 2 times), including the `Processor` features, and the usefulness of the existing examples (mentioned 2 times). A summary of analysing QUESI and NASA-TLX (evaluated as Raw TLX, without applying the weighting process) is depicted in Figure 7.6 and Figure 7.7, respectively.

Discussion

The primary goal of the conducted explorative study is the identification of methods that are reasonable for assessing the API usability of IRIS platforms. The whole process is thus taken as a basis for discussing best practices, rather than statistically analysing them.

The most prominent issue of the conducted evaluation process is the number of participants. 14 respectively 17 participants may be low, however, considering the effort that is demanded from the participants, these numbers are reasonable. Conducting the evaluation process in the context of a practical exercise is a good means to acquire participants willing to

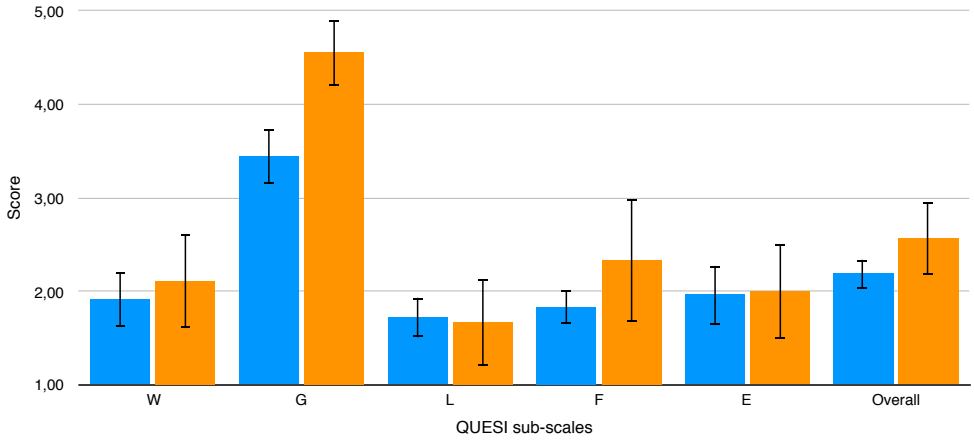

Figure 7.6. QUESI score means with 95% confidence intervals, representing results of *ML*15 (blue) and of the *MMI*16 (orange). The QUESI sub-scales are abbreviated as follows: subjective mental Workload (W), perceived achievement of Goals (G), perceived effort of Learning (L), Familiarity (F), and perceived Error rate (E). Higher values are better.

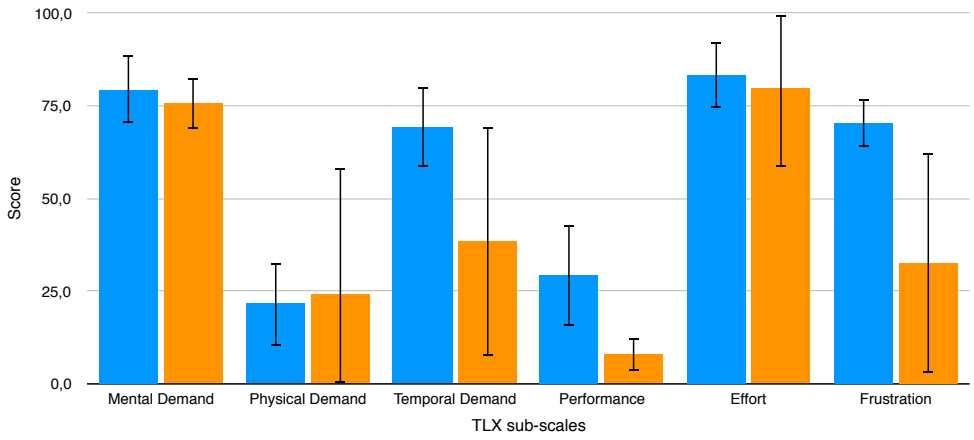

Figure 7.7. Raw TLX score means with 95% confidence intervals, representing results of the *ML*15 (blue) and of the *MMI*16 (orange).

take this high effort nevertheless. The response rate for the *ML*15 course is good. For *MMI*16, however, it is very low. This execrates the elimination of confounding factors, lowers validity, and makes these particular results almost un-exploitable.

A related issue is that it is not evident that all reported experiences completely originate from the system's API qualities. Although explicitly emphasized at the beginning of the survey not to do so, participants may have considered problems they had with the programming language Scala, the utilized math library, RIS- as well as multimodal processing concepts in general, and the teaching quality. Such confounding factors can be eliminated by keeping them as equally as possible between two conditions that are then compared, besides acquiring higher numbers of participants. With respect to an API usability study for an IRIS, this ideally implies that a second IRIS written in Scala, utilizing the same underlying concepts, had to be used during the same term to implement the programming task one more time. This can be feasible for comparing two API versions of the same IRIS. For comparing APIs of different IRIS, it may only be feasible by accepting compromises.

Lastly, it is not evident how well the chosen questionaries assess API usability in the described process. For instance, NASA-TLX is designed to be applied during or immediately after performing a task (Hart, 2006) and QUESI to assess end-user interfaces (Naumann & Hurtienne, 2010). Yet, developing software with the aid of dedicated tools and APIs is a form of using software provided by other parties via a human-computer interface too.

The secondary goal of the conducted explorative study is the potential generation of hypothesis about platform aspects that influence this software quality. The analysis of the qualitative free form responses indicates such hypothesis. For instance, that the *decoupling by semantics* technique improves the API usability vs. using lower-level APIs to implement processing steps and that code examples improve API learnability (a sub-aspect of API usability) by complementing documentation. Moreover, the analyzed responses suggest that more general documentation and more code examples should be provided.

Finally, the quantitate measures can be reviewed to get an intuition of the usability of the reference implementation's API. The QUESI results show moderate to good assessment of the perceived achievement of goals and poor values for the other sub-scales. The NASA-TLX results show a high mental demand, temporal demand, effort, and frustration as well as low physical demand and low satisfaction with the subjective performance. Compared to the long-term use of Simulator X presented by Wiebusch (2016, pp. 199–204), these results are inferior. An interpretation, however, is delicate due to the above mentioned issues.

Conclusion

Altogether, conducting usability evaluations of IRIS APIs within the context of practical exercises to master level courses using questionnaires based on programming tasks (as proposed by Piccioni et al., 2013) can be promoted as best practice, since it is a good means to acquire participants willing to take the high effort nevertheless.

The utilized survey, however, was too long. It resulted in low response rates, including partly filled in surveys that had to be skipped, and potentially low diligence during survey completion. The NASA-TLX and QUESI questionnaires on their own are good candidates when it comes to length. Their combination with additional questions, though, should be reconsidered for a full study. In addition, it should be considered to mix in methods dedicated for accessing API usability (as presented in section 3.1).

Furthermore, the study should be designed to have participants use identical hard- and software environments, e.g., in computers labs, if existent. This reduces the supervising effort and further eliminates confounding factors.

In terms of task size, a breakdown into multiple milestone that can be evaluated individually may be beneficial to likewise eliminate confounding factors and to shorten the time between task performance and assessment. In addition, aspects of the API to be evaluated should be narrowed down as closely as possible.

Finally, the effort of an IRIS API usability study should not be underestimated. Conducting it in accompany of two successive master level courses, to be able to revise the API and validate improvements, requires at least to plan one year in advance. Supervising a pratical exercise that basis on a research software platform implies technical support for a diversity of issues that are raised due to its usage by several potentially unexperienced developers. If those insights are exploited, i.e., bugs are fixed, common misconceptions are considered for API revisions, and frequently demanded documentation is improved, the research software platform is also enabled to reach a certain maturity.

7.4 Informal Insights

During the six years of development and throughout the various research- and student projects as well as utilizations for teaching certain conclusions became apparent amongst involved researches. These informal insights are summarized as four *lessons learned* (following Kuck et al., 2008; Steed, 2008). They are particularly related to API usability and are ment to support future development (revised from Fischbach et al., 2017):

Good workflow, tool support, and IDE integration are essential.
This insight may improve the efficiency of development and prevents misuse leading to poor maintainability. It is key especially for realizing the code from semantics technique. With respect to the development of Simulator X, the completion of the Protégé plugin marked the actual beginning of a utilization *as intended* of the code from semantics technique by non-core developers. However, the necessary context switches, from the IDE to Protégé, still seem to be a too high barrier. They disrupt the workflow so much that existing OWL definitions are oftentimes rather missused than fitting ones are defined. Further simplifications, e.g., an integration of dedicated ontology editing into an IDE, thus seem to be a good measure to attempt in the future.

The functional paradigm fosters a concise API that is tricky at first but beneficial when used longer and for non-trivial tasks.
Visual programming and the utilization of other popular paradigms are valid approaches to ease the the entry into RIS development. They are especially exploited by commercial products. However, these approaches are not able to completely hide all difficulties inherent to (I)RIS development. If projects reach a certain complexity they may even be hindering (cf. Wiebusch, 2016, pp. 205–206). It thus pays of in the long-term to accept a certain familiarisation phase at the beginning.

Good code examples greatly ease the issue of missing documentation.
In research, ressources for improving the documentation of a software prototype are probably even more limited than they are in economy. If prototypes have to be continuously advanced, APIs hence continuously to be refined, and documentation to be updated, the issue is exacerbated. Taking the time to implement comprehensible examples, which are required anyway for testing purposes, has proven to be a good tradeoff. This measure should be supplemented by the utilization of a management platfom for software projects that eases the contribution and access to related ressources. For instance, by granting students access to a corresponding issue tracker and by forcing relevant questions and bugs to be raised and answered there, a searchable FAQ collection can be gradually created. The overhead is minimal, since such issues have to be answered anyway, if the prototype is basis for a pratical exercise or project.

API oversimplification will result in decreased understandability.
Scala's syntax allows to hide many definitions, e.g., using the `implicit`- and type inference mechanism. Moreover, the _ wildcard allows to shorten lambda expressions by removing the need to use named variables. These mechanisms allow developers to write truly concise code. Within implementations, they should be used with caution to not yield code that is incomprehensible, even by its author after a few weeks. Within APIs, they should be used consistently, based on a concept of what is to be made explicit to the API's user and what not.

7.5 Summary

This chapter presented the results of all activities conducted to validate the proposed techniques and their reference implementation as well as to explore the assessment and improvement of IRIS maintainability. Four expert reviews evaluate the achieved decoupling and confirm the improvement of maintainability. They are strengthened in significance through their publication in peer-reviewed international workshops and conferences. This theoretical assessment is complemented by multiple proof of concept prototypes that practically validate feasibility. Two explorative studies analyse methodologies for long-term API usability assessment and yield helpful best practices for future full studies. Finally, informal insights gathered throughout the design and development are summarized as *lessons learned* to support future development in the area.

Chapter 8

Conclusion

This chapter concludes the thesis by summarizing the research motivation as well as the results and by pointing out future directions that promise to benefit form the achievements.

8.1 Summary

MMIs are a promising alternative human-computer interaction paradigm, especially if interactions have to be spatially and temporally grounded with an environment in which the user is (physically) situated. IRISs that realize MMIs for situated interaction environments, however, suffer from the *coupling dilemma*—a requirement contradiction that entails low maintainability in the short-term and hindered scientific progress in the long-term. This thesis thus researched what software techniques foster the maintainability of such systems guided by five objectives (O_{1-5}).

Based on an comprehensive review of related contributions and with the aid of an interaction use case, *fundamental (I)RIS requirements* have been identified (see chapter 2 and chapter 3): an execution model, a communication scheme, a state representation- and behavior representation model, a state- and behavior access scheme, performance, as well as maintainability, including modularity, modifiability, reusability, and API usability. A breakdown of the combination of RISs and MMSs identified the issues of independant MMS usage and concluded that an integrated MMS usage as well as the established ECS pattern are promising concepts to build on (see chapter 4). Jointly, these analyses fulfilled the first objective (O_1).

Six semantics-based techniques that extend the ECS pattern are proposed in chapter 5 to meet the fundamental requirements, as the main contribution of this thesis: semantic grounding, a semantic entity-component state, grounded actions, semantic queries, code from semantics, and decoupling by semantics. The techniques solve the ECS pattern's runtime type deficit, improve component granularity, facilitate access to entity properties outside a subsystem's component association, incorporate a concept to semantically describe behav-

ior as complement to the state representation, and enable compatibility even between IRISs. Ultimately, the six techniques constitute a solution to the *coupling dilemma* while being beneficial for multimodal processing (O_2). The guiding use case consists of an instruction-based, intentional interface. The six techniques, however, are not limited to this scenario. Access to a state- and behavior representation based on semantic descriptions as well as planning and reasoning capabilities may likewise facilitate the analysis of unintentional non-verbal communication and the generation of multimodal output.

The six techniques are the furthest abstracted results of the thesis and hence those with the highest value of re-utilization. They where developed to improve maintainability for multimodal IRISs and turned out to foster this software quality for RIS in general. They are independent of a specific programming language. Other researchers can benefit by integrating the techniques in their own software or use them to possibly enhance third-party systems.

As a secondary contribution, the reference implementation Simulator X was presented in chapter 6. It validates the feasibility of the six techniques and may be (re)used by other researchers, due to its availability under an open-source licence Wiebusch et al., 2016. The reference implementation constitutes one of two fundamental approaches for benefiting from the semantics-based techniques: low-level system integration. Alternatively, (partly) closed-source systems can be loosely coupled to a system like Simulator X and still exploit some of the advantages (cf. Eckstein, Lugrin, Wiebusch, & Latoschik, 2016; Fischbach, Wiebusch, Rehfeld, et al., 2016; Wiebusch et al., under review, 2017).

The reference implementation is based three design decision: the actor model, the web ontology language OWL, an the programming language Scala. Furthermore, it applies several implementation mechanisms to realize the proposed techniques: scala's `object` syntax, currying, implicit parameters, state variables, traits, context binding, delimited continuations, singleton actors, tool support, process integration, semantic entity references, an internal DSL, local histories, and interpolation. Both, taken design decisions and applied implementation mechanisms, are means to meet the identified functional requirements while fostering maintainability. The actor model provides a scalable communication-, execution-, and distribution scheme, fostering modularity and reusability. The choice of OWL guarantees compatibility to existing ontologies and tools. Scala facilitates realizing concise APIs (comprising internal DSL) despite the extensive requirement of callbacks. Furthermore, it is platfom independent and compatible with a large number of existing software libraries. The emphasized key mechanisms serve two kinds of purposes. Firstly, state variables, semantic entity references, and local storage of histories realize or extend functionality, conceptually demanded by the six techniques. Secondly, other key mechanisms foster API usability. Traits, implicit parameters, and Scala's `object` syntax are utilized to facilitate the use of the core API. The in-

ternal DSL makes definitions read almost as natural language descriptions, while exploiting IDE support, like autocompletion and static code analysis. API usability is further fostered by reducing the development effort with the aid of tool support and development process integration.

The reference implementation includes a feasibility validation for multimodal processing. A repertoire of common multimodal input processing steps is adding up to the integrated *miPro* framework, enabling the creation of multimodal interfaces. This application of the semantics-based techniques showcases their particular adequacy for multimodal processing, i.e., improved state access flexibility, a consistent behavior representation, compatibility with common reasoning and planning approaches, and the support of high-level APIs. *miPro* comprises two reference implementations of decision-level fusion methods: one based on tATNs and one based on unification. The integrated framework thus completes the realization of identified requirements, by providing means to check multimodal inputs with respect to its syntactic-, temporal-, and semantic correctness.

Altogether, Simulator X closes a lack of IRIS platforms with explicit MMI support (O_3). To the author's best knowledge, it is currently the only such platfom whose source code can be obtained and that is running on current hardware platforms (cf. Table 3.3 and Table 3.4).

The six semantic-based techniques as well as the reference implementation have been assessed by applying all main methods commonly applied in literature (see chapter 7). Four expert reviews have been conducted to evaluate functional and non-functional software quality aspects (O_4). The expert reviews analyse the achieved decoupling and confirm the improvement of maintainability. They are strengthened in significance through their publication in peer-reviewed international workshops, conferences, and journals. This theoretical assessment is complemented by multiple proof of concept prototypes that practically validate feasibility (O_5). In addition, two pre-studies have been conducted to explore the assessment and improvement of IRIS maintainability and to counter the lack of effective methods identified in section 3.1. Finally, informal insights gathered throughout the design and development are summarized as *lessons learned* to support future development in the area.

The presented semantic-based techniques and Simulator X have been iteratively developed for six years. During this time the reference implementation has been utilized for teaching and various research- and student projects. Many of them are still ongoing (Fischbach, Zimmerer, Link, et al., 2016; Eckstein & B. Lugrin, 2016a, 2016b; Zimmerer et al., 2016). The conceptual results of this thesis are is certainly not the perfect solution for all IRIS issues. However, they reveal novel approaches that solve serious pending issues and that may contribute to enhance future IRIS architectures. Right now, the reference implementation enables to push the IRIS-using field by providing an available technical basis that facilitates the

utilization of AI methods and in particular of multimodal processing. Research is thus facilitated beyond the idiosyncratic combination of game engines and independent processing, analysis, and simulation frameworks.

8.2 Future Work

Relating to the achievements of this thesis, this final section points out future research directions that either have been left unattended by the presented contributions or that are enabled by them.

The strictness and magnitude of real-time constraints varies amongst concrete application areas. They should be highest in areas where security is essential, e.g., in HRI, and high where synchronization between real- and virtual artefacts or immersion is crucial, e.g., in AR and VR. However, they are oftentimes not given the necessary attention. Azuma (1997) stated this circumstance twenty years ago; a fundamentally still valid criticism. The decision to use Scala as target language was beneficial for rapidly prototyping and exploring API styles. However, its dependency on the JVM and the corresponding memory menagement may negatively influence performance. This issue is researched by Stauffert et al. (2016), Rehfeld (2017), who likewise conclude that more profound insights as well as supportive tools seem necessary.

The utilized internal DSL, the tool support for the code from semantics technique, and the explorative studies presented in section 7.3 all constitue measures that have been applied to improve and assess API usability. Furthermore, Simulator X is and has been used in many projects and courses. Yet, getting familiar with this research platform takes some effort. Consequently, API usability improvements should be continued to further facilitate the utilization of the reference implementation. This implies the implementation of identified barriers that disrupt workflow and keep developers from using the system as intended. For instance, by cleverly reusing further tools to close the usability gap to the convenience that developers are used to by commercial system, especially in terms of content creation (cf. Wiebusch et al., under review, 2017). A full API usability study, planned base on the insights of the explorative studies, would also contribute to complement the already identified issues.

The integrated *miPro* framework comprises a repertoire of multimodal processing steps, including two decision-level fusion methods. For many multimodal interface realizations, both, the tATN-based implementation and the unification-based implementation, could be used. Practical evidence for what method to choose in general or under specific conditions is largely unavailable. Simulator X, however, facilitates modifications like exchanging fusion methods with one another and requires minimal changes to the surrounding application. Simulator X and miPro thus constitute an ideal testbed for the comparison and evaluation

of fusion methods—a necessary requirement for future research in the field of multimodal processing Dumas, Lalanne, and Oviatt, 2009; Lalanne et al., 2009.

Finally, the contributions of this theses certainly enable research in fields that do not develop system architectures but use IRIS to build applications. This is especially significant, since Simulator X is currently the only available platform of its kind. Being a RIS, Simulator X can be utilized in a variety of application areas (see Figure 1.3). Areas that rely on situated interaction environments, MMIs, and the application of AI methods will benefit the most. Promising directions that are already actively researched are the exploitation of the code from semantics technique for implementing social robots in the context of smart homes (Eckstein & B. Lugrin, 2016b), the utilization of multimodal interfaces for interactive surfaces (Fischbach, Zimmerer, Link, et al., 2016), and the augmentation of existing applications (not built using Simulator X) with MMIs, by carrying on the inter-RIS compatibility concept (Latoschik et al., 2016; Wiebusch et al., under review, 2017). Beyond that Simulator X and *miPro* could be utilized to generate multimodal output, e.g., taking up the ideas of the *virtuelle Werkstatt* and the virtual agent *Max* (Latoschik, 2005), and to consider unintentional, non-verbal input, e.g., to support the analysis of intentional input (following the idea of Freigang & Kopp, 2015).

Bibliography

Alatalo, T. (2011, September). An entity-component model for extensible virtual worlds. *IEEE Internet Computing*, *15*(5), 30–37.

Allard, J., Gouranton, V., Lecointre, L., Melin, E., & Raffin, B. (2002). Net Juggler: running VR juggler with multiple displays on a commodity component cluster. In *Proceedings IEEE Virtual Reality 2002* (pp. 273–274).

Allard, J., Gouranton, V., Lecointre, L., Limet, S., Melin, E., Raffin, B., & Robert, S. (2004). FlowVR: a middleware for large scale virtual reality applications. In M. Danelutto, M. Vanneschi, & D. Laforenza (Eds.), *Euro-par 2004: euro-par 2004 parallel processing* (Vol. 3149, pp. 497–505). Lecture Notes in Computer Science. Berlin, Heidelberg: Springer.

Ameri Ekhtiarabadi, A., Akan, B., Çürüklu, B., & Asplund, L. (2011). A general framework for incremental processing of multimodal inputs. In *Proceedings of the 13th International Conference on Multimodal Interfaces* (pp. 225–228). ICMI '11. Alicante, Spain: ACM.

Anastassakis, G. & Panayiotopoulos, T. (2012, June). A unified model for representing objects with physical properties, semantics and functionality in virtual environments. *Intelligent Decision Technologies*, *6*(2), 123–137.

Atrey, P. K., Hossain, M. A., El Saddik, A., & Kankanhalli, M. S. (2010). Multimodal fusion for multimedia analysis: a survey. *Multimedia Systems*, *16*(6), 345–379.

Azuma, R. T. (1997, August). A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, *6*(4), 355–385.

Bangalore, S. & Johnston, M. (2009, September). Robust understanding in multimodal interfaces. *Computational Linguistics*, *35*(3), 345–397.

Behr, J., Bockholt, U., & Fellner, D. (2011). Instantreality—A framework for industrial augmented and virtual reality applications. In D. Ma, X. Fan, J. Gausemeier, & M. Grafe (Eds.), *Virtual Reality & Augmented Reality in Industry: The 2nd Sino-German Workshop* (pp. 91–99). Berlin, Heidelberg: Springer Berlin Heidelberg.

Bergmann, K., Kahl, S., & Kopp, S. (2014). How is information distributed across speech and gesture? A cognitive modeling approach. *Cognitive Processing*, *15*(1: Special Issue: Proceedings of KogWis 2014), S84–S87.

Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., & Cruz-Neira, C. (2001, March). VR Juggler: a virtual platform for virtual reality application development. In *Proceedings IEEE Virtual Reality 2001* (pp. 89–96).

Bohus, D. (2014, January). Situated interaction: opportunities and challenges. In *Proceedings of 5th International Workshop on Spoken Dialog Systems*. Napa, California, USA.

Bolt, R. A. (1980, July). "Put-that-there": Voice and gesture at the graphics interface. *SIGGRAPH Computer Graphics*, *14*(3), 262–270.

Bower, T. (1974). The evolution of sensory systems. *Perception: Essays in honor of James J. Gibson*, 141–152.

Brooks, F. P., Jr. (1987). Walkthrough—a dynamic graphics system for simulating virtual buildings. In *Proceedings of the 1986 workshop on interactive 3d graphics* (pp. 9–21). I3D '86. Chapel Hill, North Carolina, USA: ACM.

Bueskens, C., Clemens, J., Eissfeller, B., Foerstner, R., Gadzicki, K., Peytavi, G. G., … Zachmann, G. (2014). Virtual reality for simulating autonomous deep-space navigation and mining. In *24th International conference on Artificial Reality and Teleexistence* (pp. 15–16). ICAT-EGVE. The Eurographics Association.

Carlsson, C. & Hagsand, O. (1993, September). DIVE A multi-user virtual reality system. In *Proceedings of IEEE Virtual Reality Annual International Symposium* (pp. 394–400).

Cavazza, M. & Palmer, I. (2000). High-level interpretation in virtual environments. *Applied Artificial Intelligence*, *14*(1), 125–144.

Chen, P. P.-S. (1976, March). The entity-relationship model—Toward a unified view of data. *ACM Transactions on Database Systems*, *1*(1), 9–36.

Cherubini, A., Passama, R., Fraisse, P., & Crosnier, A. (2015). A unified multimodal control framework for human-robot interaction. *Robotics and Autonomous Systems*, *70*, 106–115.

Chevaillier, P., Trinh, T. H., Barange, M., Loor, P. D., Devillers, F., Soler, J., & Querrec, R. (2012, March). Semantic modeling of virtual environments using MASCARET. In *5th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)* (pp. 1–8). IEEE Computer Society.

Clarke, S. (2004). Measuring API usability. *Doctor Dobbs Journal*, *29*(5), S1–S5.

Cohen, P. R., Johnston, M., McGee, D., Oviatt, S., Pittman, J., Smith, I., … Clow, J. (1997). Quickset: multimodal interaction for distributed applications. In *Proceedings of the Fifth ACM International Conference on Multimedia* (pp. 31–40). MULTIMEDIA '97. Seattle, Washington, USA: ACM.

Coleman, D., Ash, D., Lowther, B., & Oman, P. (1994, August). Using metrics to evaluate software system maintainability. *Computer*, *27*(8), 44–49.

Coutaz, J., Nigay, L., Salber, D., Blandford, A., May, J., & Young, R. M. (1995). Four easy pieces for assessing the usability of multimodal interaction: the care properties. In K. Nordby, P. Helmersen, D. J. Gilmore, & S. A. Arnesen (Eds.), *Human-computer interaction: interact '95* (pp. 115–120). Boston, MA: Springer US.

Cruz-Neira, C., Bierbaum, A., Hartling, P., Just, C., & Meinert, K. (2002). VR Juggler—An open source platform for virtual reality applications. In *40th AIAA Aerospace Sciences Meeting & Exhibit* (p. 754). American Institute of Aeronautics and Astronautics.

Cruz-Neira, C., Sandin, D. J., DeFanti, T. A., Kenyon, R. V., & Hart, J. C. (1992, June). The CAVE: Audio visual experience automatic virtual environment. *Communication of the ACM*, *35*(6), 64–72.

Daughtry, J. M., Farooq, U., Myers, B. A., & Stylos, J. (2009, July). API usability: report on special interest group at CHI. *ACM SIGSOFT Software Engineering Notes*, *34*(4), 27–29.

Dumas, B., Lalanne, D., & Ingold, R. (2009). HephaisTK: a toolkit for rapid prototyping of multimodal interfaces. In *Proceedings of the 2009 International Conference on Multimodal Interfaces* (pp. 231–232). ICMI-MLMI '09. Cambridge, Massachusetts, USA: ACM.

Dumas, B., Lalanne, D., & Ingold, R. (2010). Description languages for multimadal interaction: a set of guidelines and ist illustration with SMUIML. *Journal on Multimodal User Interfaces*, *3*(3), 237–247.

Dumas, B., Lalanne, D., & Oviatt, S. (2009). Multimodal interfaces: a survey of principles, models and frameworks. In D. Lalanne & J. Kohlas (Eds.), *Human machine interaction: research results of the mmi program* (pp. 3–26). Berlin, Heidelberg: Springer Berlin Heidelberg.

Eckstein, B. & Lugrin, B. (2016a). Augmented reasoning in the mirror world. In *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology* (pp. 313–314). VRST '16. Munich, Germany: ACM.

Eckstein, B. & Lugrin, B. (2016b). Dynamic context integration through modularized ontologies and semantic blueprints. In *5th international workshop on Human-Agent Interaction Design and Models*. HAIDM 2016.

Eckstein, B., Lugrin, J. L., Wiebusch, D., & Latoschik, M. E. (2016). PEARS: Physics extension and representation through semantics. *IEEE Transactions on Computational Intelligence and AI in Games*, *8*(2), 178–189.

Farooq, U. & Zirkler, D. (2010). API peer reviews: a method for evaluating usability of application programming interfaces. In *Proceedings of the 2010 ACM Conference on*

Computer Supported Cooperative Work (pp. 207–210). CSCW '10. Savannah, Georgia, USA: ACM.

Feiner, S., MacIntyre, B., Haupt, M., & Solomon, E. (1993). Windows on the world: 2D windows for 3D augmented reality. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology* (pp. 145–155). UIST '93. Atlanta, Georgia, USA: ACM.

Fischbach, M. (2015). Software techniques for multimodal input processing in realtime interactive systems. In *Proceedings of the 2015 ACM on International Conference on Multimodal Interaction* (pp. 623–627). ICMI '15. Seattle, Washington, USA: ACM.

Fischbach, M., Latoschik, M. E., Bruder, G., & Steinicke, F. (2012). smARTbox: out-of-the-box technologies for interactive art and exhibition. In *Proceedings of the 2012 virtual Reality International Conference* (19:1–19:7). VRIC '12. Laval, France: ACM.

Fischbach, M., Neff, M., Pelzer, I., Lugrin, J.-L., & Latoschik, M. E. (2013). Input device adequacy for multimodal and bimanual object manipulation in virtual environments. In *Virtuelle und Erweiterte Realität, 10. Workshop der GI-Fachgruppe VR/AR* (pp. 145–156). Informatik. Shaker Verlag.

Fischbach, M., Treffs, C., Cyborra, D., Strehler, A., Wedler, T., Bruder, G., … Steinicke, F. (2012). A mixed reality space for tangible user interaction. In *Virtuelle und Erweiterte Realität, 9. Workshop der GI-Fachgruppe VR/AR* (pp. 25–36). Informatik. Shaker Verlag.

Fischbach, M., Wiebusch, D., Giebler-Schubert, A., Latoschik, M. E., Rehfeld, S., & Tramberend, H. (2011, March). Sixton's curse – Simulator X demonstration. In *2011 IEEE Virtual Reality Conference* (pp. 255–256).

Fischbach, M., Wiebusch, D., & Latoschik, M. E. (2016, March). Semantics-based software techniques for maintainable multimodal input processing in real-time interactive systems. In *9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)* (pp. 1–6). IEEE Computer Society.

Fischbach, M., Wiebusch, D., & Latoschik, M. E. (2017, April). Semantic entity-component state management techniques to enhance software quality for multimodal VR-systems. *IEEE Transactions on Visualization and Computer Graphics*, *23*(4), 1342–1351.

Fischbach, M., Wiebusch, D., Latoschik, M. E., Bruder, G., & Steinicke, F. (2012a). Blending real and virtual worlds using self-reflection and fiducials. In *Proceedings of the 11th International Conference on Entertainment Computing* (pp. 465–468). ICEC'12. Bremen, Germany: Springer-Verlag.

Fischbach, M., Wiebusch, D., Latoschik, M. E., Bruder, G., & Steinicke, F. (2012b). smARTbox A portable setup for intelligent interactive applications. In H. Reiterer & O. Deussen

(Eds.), *Mensch & Computer 2012 — Workshopband: interaktiv informiert — allgegenwärtig und allumfassend!?* (pp. 521–524). München: Oldenbourg Verlag.

Fischbach, M., Zimmerer, C., Giebler-Schubert, A., & Latoschik, M. E. (2014, September). [DEMO] Exploring multimodal interaction techniques for a mixed reality digital surface. In *2014 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)* (pp. 335–336).

Fisher, S. S., McGreevy, M., Humphries, J., & Robinett, W. (1987). Virtual environment display system. In *Proceedings of the 1986 Workshop on Interactive 3D Graphics* (pp. 77–87). I3D '86. Chapel Hill, North Carolina, USA: ACM.

Frécon, E. (2004). *DIVE on the Internet* (Doctoral dissertation, University of Göteborg).

Freigang, F. & Kopp, S. (2015). Analysing the modifying functions of gesture in multimodal utterances. In *Proceedings of the 4th Conference on Gesture and Speech in Interaction (GESPIN)*. Nantes, France.

Fröhlich, C. (2014). *Semantische Modellierung virtueller Umgebungen auf Basis einer modularen Simulationsarchitektur* (Doctoral dissertation, Bielefeld University).

Geiger, C., Paelke, V., Reimann, C., & Rosenbach, W. (2000). A framework for the structured design of VR/AR content. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (pp. 75–82). VRST '00. Seoul, Korea: ACM.

Giebler-Schubert, A., Zimmerer, C., Wedler, T., Fischbach, M., & Latoschik, M. E. (2013). Ein digitales Tabletop-Rollenspiel für Mixed-Reality-Interaktionstechniken. In *Virtuelle und Erweiterte Realität, 10. Workshop der GI-Fachgruppe VR/AR* (pp. 181–184). Informatik. Shaker Verlag.

Glimm, B., Horrocks, I., Motik, B., Stoilos, G., & Wang, Z. (2014). HermiT: an OWL 2 reasoner. *Journal of Automated Reasoning*, *53*(3), 245–269.

Hall, D. L. & Llinas, J. (1997, January). An introduction to multisensor data fusion. *Proceedings of the IEEE*, *85*(1), 6–23.

Hart, S. G. (2006). Nasa-task load index (NASA-TLX); 20 years later. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, *50*(9), 904–908.

Heaton, J. (2015). Encog: Library of interchangeable machine learning models for Java and C#. *Journal of Machine Learning Research*, *16*, 1243–1247.

Heitlager, I., Kuipers, T., & Visser, J. (2007, September). A practical model for measuring maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)* (pp. 30–39).

Hewitt, C., Bishop, P., & Steiger, R. (1973). A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial*

Intelligence (pp. 235–245). IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc.

Horridge, M. & Bechhofer, S. (2011, January). The OWL API: a Java API for OWL ontologies. *Semantic Web*, *2*(1), 11–21.

Hoste, L., Dumas, B., & Signer, B. (2011). Mudra: a unified multimodal interaction framework. In *Proceedings of the 13th International Conference on Multimodal Interfaces* (pp. 97–104). ICMI '11. Alicante, Spain: ACM.

ISO. (2011). *Systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – system and software quality models* (ISO No. ISO/IEC 25010:2011). International Organization for Standardization. Geneva, Switzerland.

Jaimes, A. & Sebe, N. (2007, October). Multimodal human-computer interaction: a survey. *Computer Vision and Image Understanding*, *108*(1-2), 116–134.

Johnston, M. (1998). Unification-based multimodal parsing. In *Proceedings of the 17th International Conference on Computational Linguistics - Volume 1* (pp. 624–630). COLING '98. Montreal, Quebec, Canada: Association for Computational Linguistics.

Johnston, M. & Bangalore, S. (2000). Finite-state multimodal parsing and understanding. In *Proceedings of the 18th Conference on Computational Linguistics - Volume 1* (pp. 369–375). COLING '00. Saarbrücken, Germany: Association for Computational Linguistics.

Johnston, M., Cohen, P. R., McGee, D., Oviatt, S. L., Pittman, J. A., & Smith, I. (1997). Unification-based multimodal integration. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics* (pp. 281–288). ACL '98. Madrid, Spain: Association for Computational Linguistics.

Kapahnke, P., Liedtke, P., Nesbigall, S., Warwas, S., & Klusch, M. (2010). ISReal: an open platform for semantic-based 3D simulations in the 3D internet. In P. F. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Z. Pan, ... B. Glimm (Eds.), *The semantic web – iswc 2010* (pp. 161–176). Lecture Notes in Computer Science. Berlin, Heidelberg: Springer.

Kapolka, A., McGregor, D., & Capps, M. (2002). A unified component framework for dynamically extensible virtual environments. In *Proceedings of the 4th International Conference on Collaborative Virtual Environments* (pp. 64–71). CVE '02. Bonn, Germany: ACM.

Koons, D. B. & Sparrell, C. J. (1994). Iconic: Speech and depictive gestures at the human-machine interface. In *Conference Companion on Human Factors in Computing Systems* (pp. 453–454). CHI '94. Boston, Massachusetts, USA: ACM.

Kopp, S., Bergmann, K., & Kahl, S. (2013). A spreading-activation model of the semantic coordination of speech and gesture. In *Proceedings of the 35th Annual Meeting of the Cognitive Science Society (CogSci 2013)* (pp. 823–828). Berlin, Germany: Cognitive Science Society.

Kopp, S., Jung, B., Leßmann, N., & Wachsmuth, I. (2003). Max - A multimodal assistant in virtual reality construction. *KI - Künstliche Intelligenz*, *4*(03), 11–17.

Kuck, R., Wind, J., Riege, K., Bogen, M., & Birlinghoven, S. (2008). Improving the AVANGO VR/AR framework—lessons learned. In *Virtuelle und Erweiterte Realität, 5. Workshop der GI-Fachgruppe VR/AR* (pp. 209–220). Informatik. Shaker Verlag.

Lalanne, D., Nigay, L., Palanque, p., Robinson, P., Vanderdonckt, J., & Ladry, J.-F. (2009). Fusion engines for multimodal input: a survey. In *Proceedings of the 2009 International Conference on Multimodal Interfaces* (pp. 153–160). ICMI-MLMI '09. Cambridge, Massachusetts, USA: ACM.

Lange, P., Weller, R., & Zachmann, G. (2016, March). Wait-free hash maps in the entity-component-system pattern for realtime interactive systems. In *9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)* (pp. 1–8). IEEE Computer Society.

Latoschik, M. E. (2001a). A general framework for multimodal interaction in virtual reality systems: PrOSA. In W. Broll & L. Schäfer (Eds.), *Proceedings of the workshop The Future of VR and AR Interfaces - Multimodal, Humanoid, Adaptive and Intelligent at IEEE Virtual Reality 2001* (Vol. 138, pp. 21–25). Yokohama, Japan: GMD.

Latoschik, M. E. (2001b). *Multimodale Interaktion in virtueller Realität am Beispiel der virtuellen Konstruktion.* Dissertationen zur künstlichen Intelligenz. Infix Akademische Verlagsgesellschaft.

Latoschik, M. E. (2002). Designing transition networks for multimodal VR-interactions using a markup language. In *Proceedings of the 4th IEEE International Conference on Multimodal Interfaces* (pp. 411–416). ICMI '02. Washington, DC, USA: IEEE Computer Society.

Latoschik, M. E. (2005). A user interface framework for multimodal VR interactions. In *Proceedings of the 7th International Conference on Multimodal Interfaces* (pp. 76–83). ICMI '05. Torento, Italy: ACM.

Latoschik, M. E. & Blach, R. (2008). Semantic modelling for virtual worlds a novel paradigm for realtime interactive systems? In *Proceedings of the 2008 ACM Symposium on Virtual Reality Software and Technology* (pp. 17–20). VRST '08. Bordeaux, France: ACM.

Latoschik, M. E. & Fischbach, M. (2014). Engineering variance: software techniques for scalable, customizable, and reusable multimodal processing. In M. Kurosu (Ed.), *Human-*

computer interaction. theories, methods, and tools. hci 2014 (Vol. 8510, pp. 308–319). Lecture Notes in Computer Science. Cham: Springer International Publishing.

Latoschik, M. E., Froehlich, C., & Wendler, A. (2006). Scene synchronization in close coupled world representations using SCIVE. *The International Journal of Virtual Reality*, *5*(3), 47–52.

Latoschik, M. E., Lugrin, J.-L., Habel, M., Roth, D., Seufert, C., & Grafe, S. (2016). Breaking bad behavior: immersive training of class room management. In *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology* (pp. 317–318). VRST '16. Munich, Germany: ACM.

Latoschik, M. E. & Tramberend, H. (2010). Short paper: engineering realtime interactive systems: coupling & cohesion of architecture mechanisms. In *Proceedings of the 16th Eurographics Conference on Virtual Environments & Second Joint Virtual Reality* (pp. 25–28). Stuttgart, Germany: The Eurographics Association.

Latoschik, M. E. & Tramberend, H. (2011, March). Simulator X: a scalable and concurrent architecture for intelligent realtime interactive systems. In *2011 IEEE Virtual Reality Conference* (pp. 171–174).

Lawson, J.-Y. L., Al-Akkad, A.-A., Vanderdonckt, J., & Macq, B. (2009). An open source workbench for prototyping multimodal interactions based on off-the-shelf heterogeneous components. In *Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (pp. 245–254). EICS '09. Pittsburgh, PA, USA: ACM.

Leßmann, N., Kopp, S., & Wachsmuth, I. (2006). Situated interaction with a virtual human - perception, action, and cognition. In G. Rickheit & I. Wachsmuth (Eds.), *Situated communication* (pp. 287–324). Trends in Linguistics. Studies and Monographs [TiLSM]. Mouton de Gruyter.

Link, S., Barkschat, B., Zimmerer, C., Fischbach, M., Wiebusch, D., Lugrin, J. L., & Latoschik, M. E. (2016, March). An intelligent multimodal mixed reality real-time strategy game. In *2016 IEEE Virtual Reality (VR)* (pp. 223–224).

Luck, M. & Aylett, R. (2000). Applying artificial intelligence to virtual reality: Intelligent virtual environments. *Applied Artificial Intelligence*, *14*(1), 3–32.

Lugrin, J. L., Zilch, D., Roth, D., Bente, G., & Latoschik, M. E. (2016, March). FaceBo: Real-time face and body tracking for faithful avatar synthesis. In *2016 IEEE Virtual Reality (VR)* (pp. 225–226).

Lukas, L., Schwägerl, F., & Latoschik, M. E. (2010). Unifikationsbasierte Sprach-Gesten Fusion für Multimodale VR/AR-Schnittstellen. In *Virtuelle und Erweiterte Realität, 7. Workshop der GI-Fachgruppe VR/AR* (pp. 145–156). Informatik. Shaker Verlag.

Mannuß, F., Hinkenjann, A., & Maiero, J. (2008). From scene graph centered to entity centered virtual environments. In *Proceedings of the IEEE Virtual Reality Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)* (pp. 37–42). Shaker Verlag.

Martínez, H. P. & Yannakakis, G. N. (2014). Deep multimodal fusion: Combining discrete events and continuous signals. In *Proceedings of the 16th International Conference on Multimodal Interaction* (pp. 34–41). ICMI '14. Istanbul, Turkey: ACM.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., … Wilkins, D. (1998). *PDDL—The Planning Domain Definition Language* (tech. rep. No. CVC TR-98-003/DCS TR-1165). Yale Center for Computational Vision and Control.

McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, *21*(5), 2146–2189.

McLellan, S. G., Roesler, A. W., Tempest, J. T., & Spinuzzi, C. I. (1998, May). Building more usable APIs. *IEEE Software*, *15*(3), 78–86.

Mendonça, H., Lawson, J.-Y. L., Vybornova, O., Macq, B., & Vanderdonckt, J. (2009). A fusion framework for multimodal interactive applications. In *Proceedings of the 2009 International Conference on Multimodal Interfaces* (pp. 161–168). ICMI-MLMI '09. Cambridge, Massachusetts, USA: ACM.

Milgram, P., Takemura, H., Utsumi, A., & Kishino, F. (1995). Augmented reality: a class of displays on the reality-virtuality continuum. *Proc. SPIE 2351, Telemanipulator and Telepresence Technologies*, 282–292.

Miller, G. A. (1995, November). WordNet: a lexical database for english. *Communication of the ACM*, *38*(11), 39–41.

Möller, A., Diewald, S., Roalter, L., & Kranz, M. (2014). Supporting mobile multimodal interaction with a rule-based framework. *CoRR*, *abs/1406.3225*.

Murphy, R. R. (1996, January). Biological and cognitive foundations of intelligent sensor fusion. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, *26*(1), 42–51.

Musen, M. A. (2015, June). The Protégé project: a look back and a look forward. *AI Matters*, *1*(4), 4–12.

Myers, B. A. & Stylos, J. (2016, May). Improving API usability. *Communication of the ACM*, *59*(6), 62–69.

Naumann, A. & Hurtienne, J. (2010). Benchmarks for intuitive interaction with mobile devices. In *Proceedings of the 12th International Conference on Human Computer In-*

teraction with Mobile Devices and Services (pp. 401–402). MobileHCI '10. Lisbon, Portugal: ACM.

Neal, J. G., Thielman, C. Y., Dobes, Z., Haller, S. M., & Shapiro, S. C. (1989). Natural language with integrated deictic and graphic gestures. In *Proceedings of the Workshop on Speech and Natural Language* (pp. 410–423). HLT '89. Cape Cod, Massachusetts: Association for Computational Linguistics.

Ngiam, J., Khosla, A., Kim, M., Nam, J., Lee, H., & Ng, A. Y. (2011). Multimodal deep learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (pp. 689–696).

Nigay, L. & Coutaz, J. (1993). A design space for multimodal systems: concurrent processing and data fusion. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems* (pp. 172–178). CHI '93. Amsterdam, The Netherlands: ACM.

Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., … Zenger, M. (2004). *An overview of the Scala programming language* (tech. rep. No. LAMP-REPORT-2006-001). École Polytechnique Fédérale de Lausanne (EPFL).

Olmedo, H., Escudero, D., & Cardenoso, V. (2008). A framework for the development of applications allowing multimodal interaction with virtual reality worlds. In *Communications papers: 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision WSCG'2008* (pp. 79–86).

Oman, P. & Hagemeister, J. (1992, November). Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992* (pp. 337–344).

Oviatt, S. (2003, September). Advances in robust multimodal interface design. *EEE Computer Graphics and Applications*, *23*(5), 62–68.

Oviatt, S. (2012). Multimodal interfaces. In J. A. Jacko (Ed.), *Human Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications, Third Edition*. Human Factors and Ergonomics. CRC Press.

Oviatt, S. & Cohen, P. R. (2015). The paradigm shift to multimodality in contemporary computer interfaces. *Synthesis Lectures On Human-Centered Informatics*, *8*(3), 1–243.

Oviatt, S., Coulston, R., & Lunsford, R. (2004). When do we interact multimodally?: Cognitive load and multimodal communication patterns. In *Proceedings of the 6th International Conference on Multimodal Interfaces* (pp. 129–136). ICMI '04. State College, PA, USA: ACM.

Parnas, D. L. (1994). Software aging. In *Proceedings of the 16th International Conference on Software Engineering* (pp. 279–287). ICSE '94. Sorrento, Italy: IEEE Computer Society Press.

Pfleger, N. (2004). Context based multimodal fusion. In *Proceedings of the 6th International Conference on Multimodal Interfaces* (pp. 265–272). ICMI '04. State College, PA, USA: ACM.

Piccioni, M., Furia, C. A., & Meyer, B. (2013, October). An empirical study of API usability. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement* (pp. 5–14).

Ponder, M., Papagiannakis, G., Molet, T., Magnenat-Thalmann, N., & Thalmann, D. (2003, July). VHD++ development framework: towards extendible, component based VR/AR simulation engine featuring advanced virtual character technologies. In *Proceedings Computer Graphics International 2003* (pp. 96–104).

Rehfeld, S. (2017). *Untersuchung der Nebenläufigkeit, Latenz und Konsistenz asynchroner Interaktiver Echtzeitsysteme mittels Profiling und Model Checking* (Doctoral dissertation, University of Würzburg). To appear.

Rehfeld, S., Latoschik, M. E., & Tramberend, H. (2016, March). Estimating latency and concurrency of asynchronous real-time interactive systems using model checking. In *2016 IEEE Virtual Reality (VR)* (pp. 57–66).

Rehfeld, S., Tramberend, H., & Latoschik, M. E. (2013, March). An actor-based distribution model for realtime interactive systems. In *Proceedings of the IEEE Virtual Reality 6th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)* (pp. 9–16).

Rehfeld, S., Tramberend, H., & Latoschik, M. E. (2014). Profiling and benchmarking event- and message-passing-based asynchronous realtime interactive systems. In *Proceedings of the 20th ACM Symposium on Virtual Reality Software and Technology* (pp. 151–159). VRST '14. Edinburgh, Scotland: ACM.

Reynolds, C. W. (1987, August). Flocks, herds and schools: a distributed behavioral model. *SIGGRAPH Computer Graphics*, *21*(4), 25–34.

Riaz, M., Mendes, E., & Tempero, E. (2009, October). A systematic review of software maintainability prediction and metrics. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement* (pp. 367–377). IEEE.

Ruiz, N., Chen, F., & Oviatt, S. (2010). Chapter 12 - Multimodal Input. In J.-P. Thiran, F. Marqués, & H. Bourlard (Eds.), *Multimodal Signal Processing* (pp. 231–255). Oxford: Academic Press.

Salem, M., Kopp, S., Wachsmuth, I., & Joublin, F. (2010, October). Generating robot gesture using a virtual agent framework. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 3592–3597).

Schmidt, A., Van de Velde, W., & Kortuem, G. (2000). Situated interaction in ubiquitous computing. In *CHI '00 Extended Abstracts on Human Factors in Computing Systems* (pp. 374–374). CHI EA '00. The Hague, The Netherlands: ACM.

Serrano, M., Nigay, L., Lawson, J.-Y. L., Ramsay, A., Murray-Smith, R., & Denef, S. (2008). The OpenInterface framework: a tool for multimodal interaction. In *CHI '08 Extended Abstracts on Human Factors in Computing Systems* (pp. 3501–3506). CHI EA '08. Florence, Italy: ACM.

Sharma, R., Pavlovic, V. I., & Huang, T. S. (1998, May). Toward multimodal human-computer interface. *Proceedings of the IEEE*, *86*(5), 853–869.

Shaw, C. & Green, M. (1993, September). The MR toolkit peers package and experiment. In *Proceedings of IEEE Virtual Reality Annual International Symposium* (pp. 463–469).

Shaw, C., Green, M., Liang, J., & Sun, Y. (1993, July). Decoupled simulation in virtual reality with the MR toolkit. *ACM Transactions on Information Systems (TOIS)*, *11*(3), 287–317.

Shen, J. & Pantic, M. (2013, December). HCI^2 Framework: A Software Framework for Multimodal Human-Computer Interaction Systems. *IEEE Transactions on Cybernetics*, *43*(6), 1593–1606.

Sire, S. & Chatty, S. (2004). The markup way to multimodal toolkits. In *W3C Workshop on Multimodal Interaction*.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., & Katz, Y. (2007). Pellet: a practical OWL-DL reasoner. *Web Semantics: science, services and agents on the World Wide Web*, *5*(2), 51–53.

Stauffert, J. P., Niebling, F., & Latoschik, M. E. (2016, March). Reducing application-stage latencies for real-time interactive systems. In *9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)* (pp. 1–7). IEEE Computer Society.

Steed, A. (2008). Some useful abstractions for re-usable virtual environment platforms. In *Proceedings of the IEEE Virtual Reality Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*. Shaker Verlag.

Stein, B. E. & Meredith, M. A. (1993). *The merging of the senses.* The MIT Press.

Streitz, N. A., Röcker, C., Prante, T., Stenzel, R., & van Alphen, D. (2003). Situated interaction with ambient information: facilitating awareness and communication in ubiquitous work environments. In D. Harris, V. Duffy, M. Smith, & C. Stephanidis (Eds.), *Human*

Centred Computing: Cognitive, Social, and Ergonomic Aspects (pp. 133–137). Lawrence Erlbaum Publishers.

Sutherland, I. E. (1964). Sketch Pad—A man-machine graphical communication system. In *Proceedings of the SHARE Design Automation Workshop* (pp. 6329–6346). DAC '64. New York, NY, USA: ACM.

Tang, W. W., Lo, K. W., Chan, A. T., Chan, S., Leong, H. V., & Ngai, G. (2011). I*Chameleon: a scalable and extensible framework for multimodal interaction. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems* (pp. 305–310). CHI EA '11. Vancouver, BC, Canada: ACM.

Taylor, R. M., II, Hudson, T. C., Seeger, A., Weber, H., Juliano, J., & Helser, A. T. (2001). VRPN: a device-independent, network-transparent VR peripheral system. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (pp. 55–61). VRST '01. Baniff, Alberta, Canada: ACM.

Taylor, R. M., Jerald, J., VanderKnyff, C., Wendt, J., Borland, D., Marshburn, D., … Whitton, M. C. (2010, April). Lessons about virtual environment software systems from 20 years of VE building. *Presence: Teleoper. Virtual Environ. 19*(2), 162–178.

Tramberend, H. (1999, March). Avocado: a distributed virtual reality framework. In *Proceedings IEEE Virtual Reality* (pp. 14–21).

Turk, M. (2014). Multimodal interaction: a review. *Pattern Recognition Letters*, *36*, 189–195.

Turk, M. & Robertson, G. (2000, March). Perceptual user interfaces (introduction). *Communication of the ACM*, *43*(3), 32–34.

W3C OWL Working Group. (2009, October). *OWL 2 Web Ontology Language Document Overview*. W3C.

Wagner, J., Lingenfelser, F., Baur, T., Damian, I., Kistler, F., & André, E. (2013). The social signal interpretation (SSI) framework: multimodal signal processing and recognition in real-time. In *Proceedings of the 21st ACM International Conference on Multimedia* (pp. 831–834). MM '13. Barcelona, Spain: ACM.

Wahlster, W. (1991). Intelligent User Interfaces. In J. W. Sullivan & S. W. Tyler (Eds.), (Chap. User and Discourse Models for Multimodal Communication, pp. 45–67). New York, NY, USA: ACM.

Wiebusch, D. (2016). *Reusability for intelligent realtime interactive systems* (Doctoral dissertation, University of Würzburg).

Wiebusch, D., Fischbach, M., Latoschik, M. E., & Tramberend, H. (2012). Evaluating scala, actors, & ontologies for intelligent realtime interactive systems. In *Proceedings of the 18th ACM Symposium on Virtual Reality Software and Technology* (pp. 153–160). VRST '12. Toronto, Ontario, Canada: ACM.

Wiebusch, D., Fischbach, M., Strehler, A., Latoschik, M. E., Bruder, G., & Steinicke, F. (2012). Evaluation von Headtracking in interaktiven virtuellen Umgebungen auf Basis der Kinect. In *Virtuelle und Erweiterte Realität, 9. Workshop der GI-Fachgruppe VR/AR* (pp. 189–200). Informatik. Shaker Verlag.

Wiebusch, D. & Latoschik, M. E. (2012, March). Enhanced decoupling of components in intelligent realtime interactive systems using ontologies. In *5th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)* (pp. 43–51). IEEE Computer Society.

Wiebusch, D., Latoschik, M. E., & Tramberend, H. (2010). Ein konfigurierbares World-Interface zur Kopplung von KI-Methoden an Interaktive Echtzeitsysteme. In *Virtuelle und Erweiterte Realität, 7. Workshop der GI-Fachgruppe VR/AR* (pp. 47–58). Informatik. Shaker Verlag.

Wiebusch, D., Zimmerer, C., & Latoschik, M. E. (under review, 2017). Cherry-picking RIS functionality – integration of game and VR engine sub-systems based on entities and events. In *10th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*. IEEE Computer Society.

Wingrave, C. A. & LaViola, J. J. (2010, April). Reflecting on the design and implementation issues of virtual environments. *Presence: Teleoper. Virtual Environ. 19*(2), 179–195.

Wu, L., Oviatt, S. L., & Cohen, P. R. (1999, December). Multimodal integration-a statistical view. *IEEE Transactions on Multimedia*, *1*(4), 334–341.

Zeltzer, D., Pieper, S., & Sturman, D. J. (1989). An integrated graphical simulation platform. In *Proceedings of Graphics Interface '89* (pp. 266–274). GI '89. London, Ontario, Canada.

Zimmerer, C., Fischbach, M., & Latoschik, M. (2014). Fusion of mixed-reality tabletop and location-based applications for pervasive games. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces* (pp. 427–430). ITS '14. Dresden, Germany: ACM.

Zimmerer, C., Fischbach, M., & Latoschik, M. E. (2016). Maintainable management and access of lexical knowledge for multimodal virtual reality interfaces. In *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology* (pp. 347–348). VRST '16. Munich, Germany: ACM.

Online Sources

AMIS group. (2017). planning4j. https://code.google.com/archive/p/planning4j/. Last accessed 2017-04-23.

Bilas , Scott. (2002). A Data-Driven Game Object System. Game Developer Conference (GDC) talk, http://gamedevs.org/uploads/data-driven-game-object-system.pdf. Last accessed 2017-04-09.

Cognitum. (2015). Fluent Editor. http://www.cognitum.eu/semantics/FluentEditor/. Last accessed 2017-04-12.

Dvorak, Martin. (2017). JBullet. http://jbullet.advel.cz. Last accessed 2017-02-17.

EPIC GAMES, INC. (2017). Unreal Engine 4. https://www.unrealengine.com. Last accessed 2017-02-17.

Fischbach, M., Wiebusch, D., Rehfeld, S., Tramberend, H., & Latoschik, M. E. (2016). Simulator X. http://www.hci.uni-wuerzburg.de/projects/simulator-x.html. Last accessed 2016-12-09.

Fischbach, M., Zimmerer, C., Link, S., Giebler-Schubert, A., & Latoschik, M. E. (2016). XRoads. http://www.hci.uni-wuerzburg.de/projects/xroads.html. Last accessed 2017-05-12.

HTC Corporation. (2017). VIVE. https://www.vive.com. Last accessed 2017-05-12.

JIDE Software. (2017). JIDE Components. http://www.jidesoft.com. Last accessed 2017-04-29.

Kaltenbrunner, M. (2017). TUIO. http://www.tuio.org. Last accessed 2017-05-05.

Kern, F., Kullman, P., Wiebusch, D., & Lugrin, J.-L. (2016). Project Results of the 3D User Interfaces Course. https://youtu.be/jkbcIXtNK5k. Last accessed 2017-02-17.

Kern, F., Kullman, P., Zöller, J., Zimmerer, C., Fischbach, M., & Latoschik, M. E. (2016). Project Results of the Multimodal Interaction Course. https://youtu.be/6yb0AwsZzdU. Last accessed 2017-02-17.

Khronos Group. (2017). COLLADA – 3D Asset Exchange Schema. https://www.khronos.org/collada/. Last accessed 2017-05-05.

Lanier, J. (1988). A vintage virtual reality interview. Originally published at Whole Earth Review, http://www.jaronlanier.com/vrint.html. Last accessed 2017-05-20.

Latoschik, M. E. (2017). IEEE VR Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS). http://www.searis.net. Last accessed 2017-02-17.

Leap Motion, Inc. (2017). Leap Motion. https://www.leapmotion.com. Last accessed 2017-05-05.

Lightbend Inc. (2016). akka. http://akka.io. Last accessed 2017-04-12.

Lightbend Inc. (2017). sbt—The interactive build tool. http://www.scala-sbt.org. Last accessed 2017-04-29.

Orchard, Leslie. (2013). Parsec Patrol Diaries: Entity Component Systems. https://blog.lmorchard.com/2013/11/27/entity-component-system/. Last accessed 2017-03-21.

Pegasus Spiele. (2014, June). Quest - Zeit der Helden. Retrieved from http://www.pegasus.de/quest-zeit-der-helden/.

Unity Technologies. (2017). Unity. http://www.unity3d.com. Last accessed 2017-02-17.

W3C. (2017). SPARQL Query Language for RDF. https://www.w3.org/TR/rdf-sparql-query/. Last accessed 2017-04-02.

West, Mick. (2007). Evolve Your Heirachy. http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/. Last accessed 2017-03-21.

Wiebusch, D., Fischbach, M., Rehfeld, S., Tramberend, H., & Latoschik, M. E. (2016). Simulator X on GitHub. https://github.com/simulator-x/simulator-x. Last accessed 2017-13-05.

Printed Sources

Link, S. (2017). *Integration of an extended unification approach into a real-time interactive system*. Master's thesis, University of Würzburg. To be published.

Seufert, A. (2013). *Das MagiCol Window – Umsetzung einer Window-to-World Mixed-Reality Umgebung für kollaborative Arbeiten an verdeckten Infrastrukturen am Beispiel von Leitungssystemen*. Bachelor's thesis, University of Würzburg.

Zimmerer, C. (2016). *Software techniques for decision-level multimodal fusion – enhancing maintainability for multimodal real-time interactive systems*. Master's thesis, University of Würzburg.

Appendix A

System Availability

Table A.1 and Table A.2 specify how available MMSs and RISs can be obtained. Authors of systems with no (web) source have been contacted per mail. Systems whose authors did not reply (within several months) are consequently listed as unavailable.

Table A.1. Sources for available MMSs as a supplement to Table 3.3.

Name	Available	Source	Reference
Put that There	no		Bolt (1980)
Cubricon	no		Neal et al. (1989)
eXpert TRAnslator	no		Wahlster (1991)
ICONIC	no		Koons and Sparrell (1994)
QuickSet	no		Cohen et al. (1997)
SGIM & virtuelle Werkstatt	no		Latoschik (2001a, 2005)
OpenInterface (OI)	yes	https://forge.openinterface.org	Serrano et al. (2008)
SKEMMI (OI)	yes	https://forge.openinterface.org	Lawson et al. (2009)
Meanings4Fusion (OI)	yes	https://kenai.com/projects/meanings4fusion	Mendonça et al. (2009)
i*Chameleon	no		Tang et al. (2011)
Mudra	no		Hoste et al. (2011)
unnamed system using COLD	no		Ameri Ekhtiarabadi et al. (2011)
HCI^2	yes	http://ibug.doc.ic.ac.uk/resources/hci2-framework/	Shen and Pantic (2013)
SSI	yes	http://hcm-lab.de/projects/ssi/download/	Wagner et al. (2013)
M3I	yes	http://www.eislab.net/m3i/	Möller et al. (2014)
unnamed system	no		Cherubini et al. (2015)
miPro (Simulator X)	yes	https://github.com/simulator-x	Latoschik and Fischbach (2014), Fischbach et al. (2017)

Table A.2. Sources for available RISs as a supplement to Table 3.4.

Name	Available	Source	Reference
Walkthrough	no		Brooks (1987)
bolio	no		Zeltzer et al. (1989)
MR Toolkit	limited	on demand (see mail)	Shaw et al. (1993)
DIVE	no		Carlsson and Hagsand (1993), Frécon (2004)
Avocado (Avango)	limited	https://github.com/vrsys/avango	Tramberend (1999)
VR Juggler	limited	https://github.com/vrjuggler/vrjuggler	Bierbaum et al. (2001), Allard et al. (2002)
I4D	no		Geiger et al. (2000)
NPSNET-V	limited	https://sourceforge.net/projects/npsnetv/	Kapolka et al. (2002)
VHD++	limited	https://sourceforge.net/projects/vhdplus/	Ponder et al. (2003)
FlowVR	yes	http://flowvr.sourceforge.net	Allard et al. (2004)
SCIVE	no		Latoschik et al. (2006), Fröhlich (2014)
AvangoNG	yes	https://github.com/vrsys/avangong	Kuck et al. (2008)
ISReal	yes	http://www.dfki.de/~klusch/isreal/html/software.html	Kapahnke et al. (2010)
instantReality	limited		Behr et al. (2011)
REVE	yes		Anastassakis and Panayiotopoulos (2012)
MASCARET	yes	http://svn.cerv.fr/trac/mascaret2	Chevaillier et al. (2012)
Simulator X	yes	https://github.com/simulator-x/simulator-x	Latoschik and Tramberend (2011), Fischbach et al. (2017)
Unreal Engine 4	yes	https://www.unrealengine.com/ue4-on-github	EPIC GAMES, INC. (2017)
Unity	limited	https://store.unity.com	Unity Technologies (2017)